移动开发经典丛书

构建图形丰富与更佳性能的原生应用



Pro Android C++ with the NDK

Android C++ 高级编程——使用NDK

[美] Onur Cinar 著
于红 佘建伟 冯艳红 译



Apress®

清华大学出版社

移动开发经典丛书

Android C++高级编程
——使用 NDK

[美] Onur Cinar 著
于红 佘建伟 冯艳红 译

清华大学出版社
北 京

Onur Cinar
Pro Android C++ with the NDK
EISBN：978-1-4302-4827-9

北京市版权局著作权合同登记号　图字：01-2013-4603

图书在版编目(CIP)数据

Android C++高级编程——使用 NDK / (美) 辛纳（Cinar,O.）著；于红，佘建伟，冯艳红　译.
—北京：清华大学出版社，2014
(移动开发经典丛书)
书名原文：Pro Android C++ with the NDK

ISBN 978-7-302-34301-1

Ⅰ. ①A…　Ⅱ. ①辛…　②于…　③佘…　④冯…　Ⅲ. ①移动终端—应用程序—程序设计
Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字(2013)第 251610 号

责任编辑：王　军　　于　平
装帧设计：牛艳敏
责任校对：邱晓玉
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张：**22.5　　**字　　数：**548 千字
版　　次：2014 年 1 月第 1 版　　**印　　次：**2014 年 1 月第 1 次印刷
印　　数：1～4000
定　　价：59.80 元

产品编号：

译 者 序

Android 是 Google 公司基于 Linux 平台开发的开源手机操作系统，自然要对 C、C++提供原生的支持。通过 Google 发布的 Android 手机 NDK(Native Development Kit)，应用程序可以非常方便地实现 Java 与 C/C++代码的相互沟通。合理地使用 NDK，可以提高应用程序的执行效率。所以，对于 Android 开发人员来说，NDK 是必须掌握的工具。

本书的作者 Onur Cinar 在美国宾州费城 Drexel 大学获得计算机科学理学学士学位。他有 17 年的移动通信领域大规模复杂软件项目的设计、开发和管理经验。自 Android 平台问世以来，Onur Cinar 一直积极从事 Android 开发工作，目前在微软 Skype 分部担任 Android 平台的 Skype 客户端高级产品工程经理。出版了多部 Android 开发应用方面的图书。

本书提供了 Android NDK 开发的全面信息，介绍了从 NDK 开发环境搭建的每一步细节，NDK 的基本概念和体系结构，具体的开发流程和方法。同时还比较详尽地介绍了 Android NDK 对 C、C++标准库的支持。是一本关于 Android NDK 开发的全新入门指南。

本书译者团队具有丰富的系统设计与开发经验。于红在计算机应用技术领域工作 20 余年，承担 Java 语言程序设计、Visual Basic 等程序设计、操作系统等课程的教学任务，熟悉程序设计类课程的教学规律，具有较强的语言组织和语言表达能力，在国内外期刊及知名国际会议上发表论文 40 余篇，其中被三大检索收录 22 篇，出版教材 3 部。佘建伟先后就职于腾讯(大连)无线研发中心和东软(大连)集团有限公司。主要从事移动互联网应用开发以及嵌入式产品的设计和开发。从 2010 年至今一直从事 Android 应用与游戏开发和 Framework 内核以及 Android 底层开发技术的研究，对 Android 内核有较为深刻的理解。此外对移动互联网产品交互设计和产品运营也有一定的研究。目前已出版一本译著《Android 4 高级编程(第 3 版)》。冯艳红从事计算机应用技术教学及研究工作 7 年多，主要承担 Java 语言程序设计、C 语言程序设计、C++面向对象的程序设计等课程的教学工作，参与了多个项目的开发，具有较强的理论基础和程序开发经验。

于红负责翻译第 3～10 章，佘建伟负责翻译第 11～14 章，冯艳红负责翻译第 1 章、第 2 章、前言、内封、封底等。参与翻译活动的还有孙庚、王芳、黄璐、史鹏辉、崔春雷、何南、孙京恩、王美妮；另外李青、孔亮亮、王宁三位同学协助完成本书部分内容的翻译，没有他们的帮助不可能完成本书的翻译工作，在此对他们表示衷心的感谢。由于时间仓促、译者的水平及精力有限，一定存在谬误，希望读者们多多批评指正。

译者

作 者 简 介



Onur Cinar 有超过 17 年的移动和通信领域大规模复杂软件项目的设计、开发和管理经验。他的专业技能包括 VoIP、视频通信、移动应用程序、网格计算和不同平台上的网络技术。从 Android 平台问世他就一直积极从事这方面的工作。他是 Apress 出版的 *Android Apps with Eclipse* 一书的作者。他在美国宾州费城 Drexel 大学获得计算机科学理学学士学位。现就职于微软 Skype 分部，任 Android 平台的 Skype 客户端高级产品工程经理。

技术审校者简介



Grant Allen 在 IT 领域工作了 20 年，主要职位是 CTO、企业架构师和数据库架构师。他曾经在全球各地的私企、学术界和政府部门工作过，专长是进行全球通用的系统设计、开发和性能优化。他经常在产业界和学术界的会议上发言，涉及的主题从数据挖掘到兼容性，以及诸如数据库(DB2、Oracle、SQL Server 以及 MySQL)，内容管理，协作，颠覆性的创新和像 Android 这样的移动生态系统之类的技术。

他的第一个 Android 应用程序是一个提醒他完成所有其他未完成的 Android 项目的任务列表。

Grant 在谷歌工作，利用空闲时间攻读博士学位，博士论文的研究题目是构建创新性的高技术环境。

他是 *Beginning DB2: From Novice to Professional* (Apress, 2008)的作者，*Oracle SQL Recipes: A Problem-Solution Approach* (Apress, 2010)和 *The Definitive Guide to SQLite, 2nd Edition* (Apress, 2010)的主编。

前　　言

Android 是移动电话市场的主要角色而且其市场份额正在持续增长。它是第一个完整的、开放的、免费的移动平台，该平台给移动应用开发者提供了无限的机会。

虽然 Android 平台的官方程序语言是 Java，但应用开发者不限于仅使用 Java 技术。

Android 允许应用开发者通过 Android 原生开发包(NDK)使用诸如 C 和 C++之类的原生代码语言实现他们的部分应用。本书中我们将学习如何用 Android NDK 通过原生代码语言去实现自己的 Android 应用中对性能要求较高的部分。

本书介绍了原生应用开发、可用的原生 API 以及故障排除技术的详细叙述，包括用按步骤的指导和屏幕截图以帮助 Android 开发人员迅速达到开发原生应用的目的。

主要内容：

- 在主要的操作系统上安装 Android 原生开发环境。
- 使用 Eclipse 集成开发环境开发原生代码。
- 使用 Java 原生接口(JNI)将原生代码与 Java 代码连接。
- 用 SWIG 自动生成 JNI 代码。
- 用 POSIX 和 Java 线程开发多线程原生应用。
- 用 POSIX sockets 开发网络原生应用。
- 用 logging、GDB 和 Eclipse 调试器调试原生代码。
- 用 Valgrind 分析内存问题。
- 用 GProf 测试应用性能。
- 用 SIMD/NEON 优化原生代码。

下载代码

读者可以在 www.apress.com 上下载本书源代码。

联系作者

读者可以通过作者的 Android C++ with the NDK 网站 http://www.zdo.com/android-c++-with-the-ndk 联系作者。

目　　录

第 1 章

Android 平台上的 C++入门

毋庸置疑，探索和实践是学习的最佳方法。本书一开始就为读者讲解功能完备的开发环境，使读者可以在以后各章的学习过程中用实例进行探索和实验。Android C++开发环境主要由以下几部分构成：

- Android 软件开发包(Software Development Kit，SDK)
- Android 原生开发包(Native Development Kit，NDK)
- Eclipse 上的 Android 开发工具(Android Development Tools，ADT)插件
- Java 开发包(Java Development Kit，JDK)
- Apache ANT 构建系统
- GNU Make 构建系统
- Eclipse IDE

本章循序渐进地讲解正确配置 Android C++开发环境的步骤，Android 开发工具可以在以下三种操作系统平台上运行：

- Microsoft Windows
- Apple Mac OS X
- Linux

由于不同操作系统的需求和安装步骤差异较大，因此下面将分别阐述不同操作系统上 Android C++开发环境的安装步骤，可以跳过你不使用的操作系统。

1.1 Microsoft Windows

Android 开发工具可以在 Windows XP(仅限于 32 位)、Vista 或 Windows 7 中运行。在本节中你需要下载并安装以下组件：

- Java JDK 6
- Apache ANT 构建系统
- Android SDK

- Cygwin
- Android NDK
- Eclipse IDE

1.1.1 在 Windows 平台上下载并安装 JDK 开发包

Android 开发工具要求必须安装 JDK(Java Development Kit)，不能只安装 JRE(Java Runtime Edition)，在安装 Android 开发工具之前需要先安装 Java JDK 6。

注意：

为遵守上述版本号，Android 开发工具只支持版本 5 或者 6 的 Java 编译器。用 JDK 6 更简单且不易出错。

Android 开发工具支持多种发行版本的 JDK，例如：IBM JDK、Open JDK 以及 Oracle JDK(即以前的 Sun JDK)。因为 Oracle JDK 支持的平台较多，本书使用 Oracle JDK 为例进行讲解。

请访问 www.oracle.com/technetwork/java/javase/downloads/index.html 网站，按照以下步骤下载 Oracle JDK：

(1) 如图 1-1 所示，单击 JDK 6 下载按钮开始下载，本书编写时最新版本的 Oracle JDK 6 是 Update 33。



图 1-1 Oracle JDK 6 下载按钮

(2) 单击 Oracle JDK 6 Download 按钮之后进入支持平台的 Oracle JDK 6 安装包清单页面。

(3) 选中 Accept License Agreement 选项并下载 Windows x86 安装包，如图 1-2 所示。

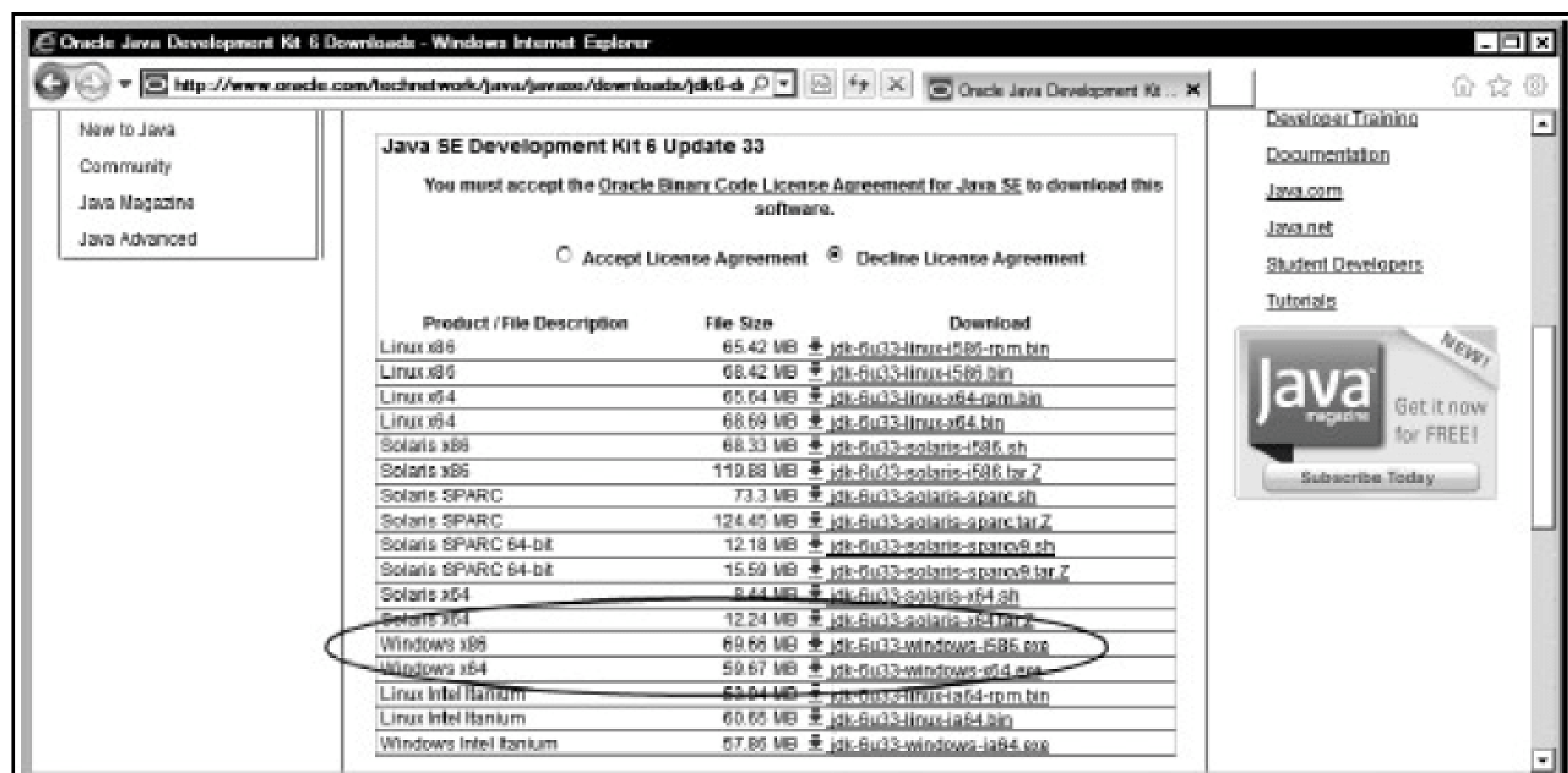


图 1-2 下载 Windows x86 下的 Oracle JDK 6

现在可以安装了。Windows 平台下的 Oracle JDK 6 安装包带有图形安装向导，它将指导你完成 JDK 的安装。安装向导首先安装 JDK，然后安装 JRE。在安装过程中，安装向导会让你指定安装目录以及要安装的组件，如图 1-3 所示。当然，此处可以使用默认值。此时，要记住 JDK 的安装目录以备环境变量设置时使用。

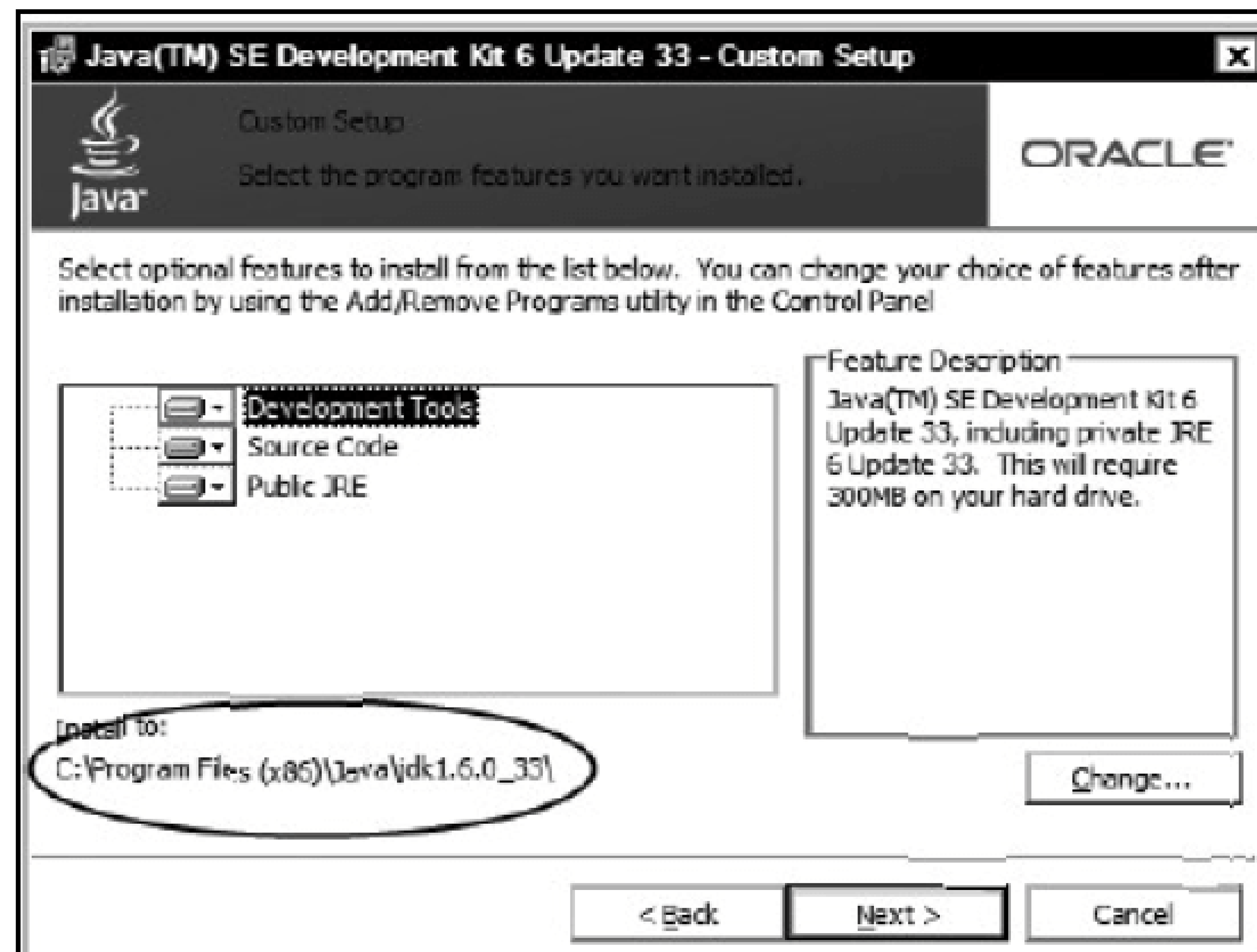


图 1-3　Oracle JDK 6 安装目录

安装完成时 JDK 准备就绪，安装向导不会自动将 Java 二进制目录加入系统可执行文件搜索路径，即 PATH 环境变量中，这一步要在 JDK 安装的最后一步手工完成：

(1) 在 Start 按钮菜单中选择 Control Panel。

(2) 单击 System 图标进入 System Properties 对话框。

(3) 单击 Advanced 选项卡，然后单击此选项卡中的 Environment Variables 按钮，如图 1-4 所示。

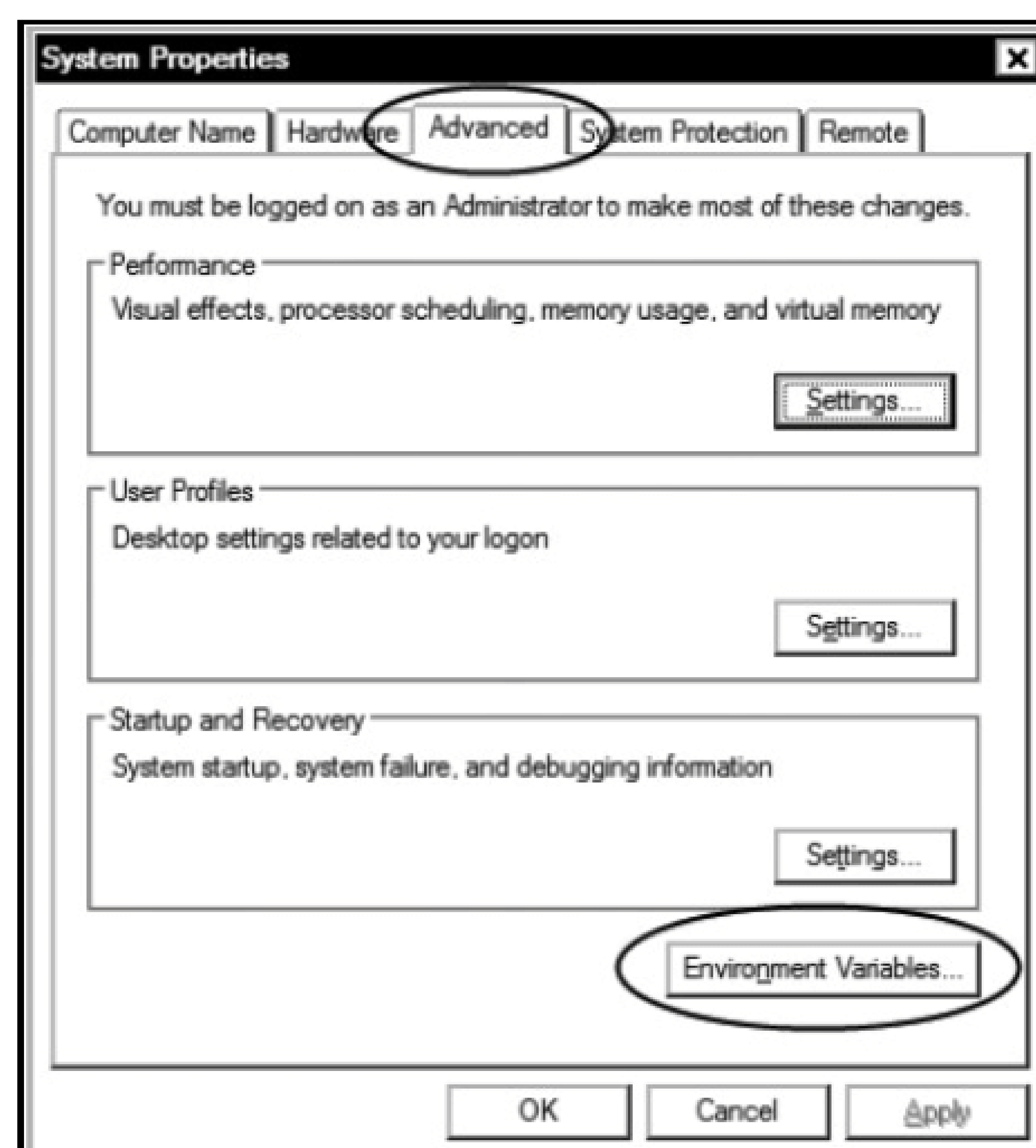


图 1-4　System Properties 对话框

(4) 单击 Environment Variables 按钮将启动 Environment Variables 对话框，该对话框由两部分构成：上面是 User Environment Variables(用户环境变量)，下面是 System Environment Variables(系统环境变量)。

(5) 在系统变量部分单击 New 按钮定义新的环境变量，如图 1-5 所示。

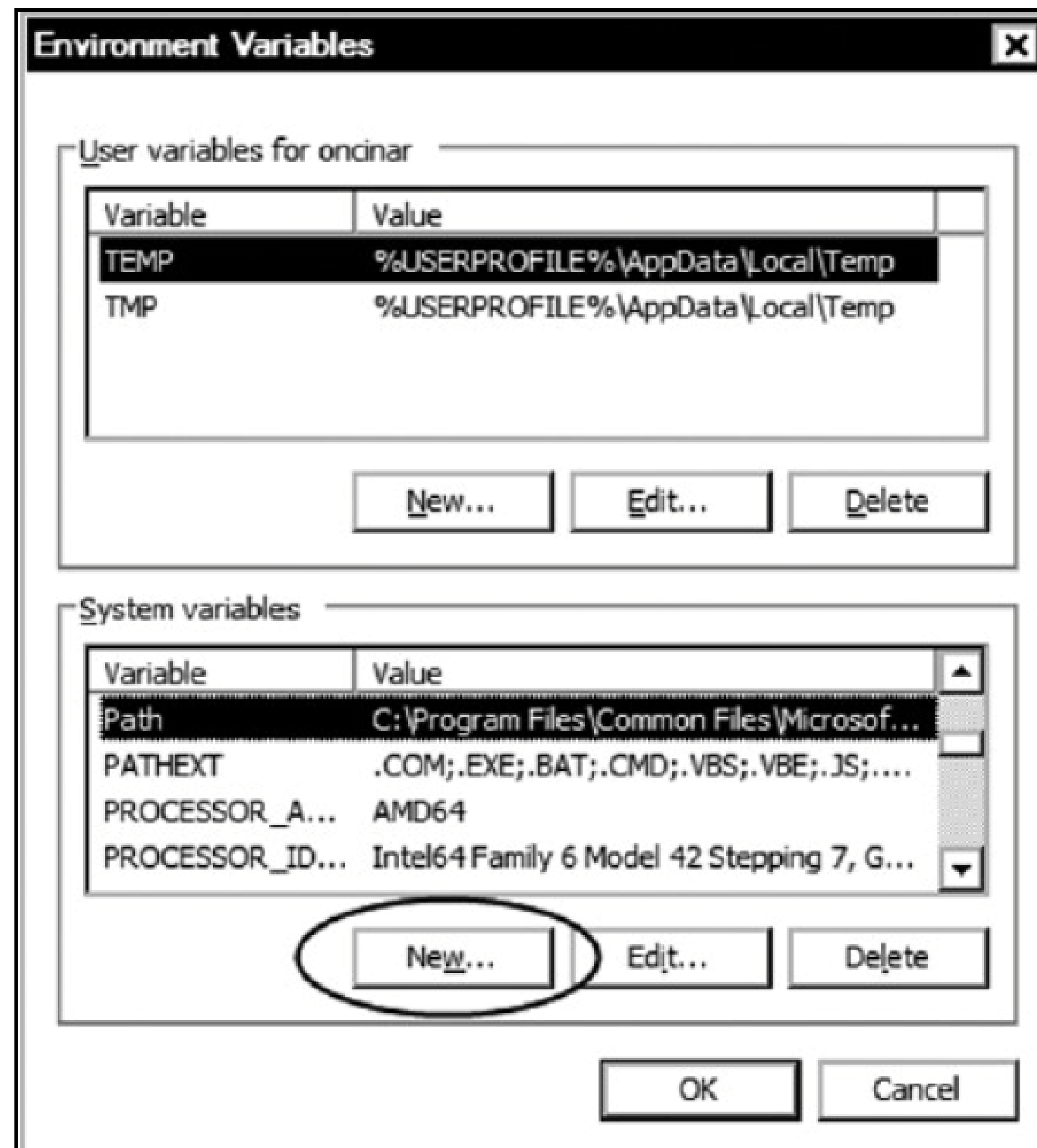


图 1-5　Environment Variables 对话框

(6) 将变量名设置成 JAVA_HOME，变量值设置成在前面的安装过程中记录下来的 Oracle JDK 的安装目录，如图 1-6 所示。

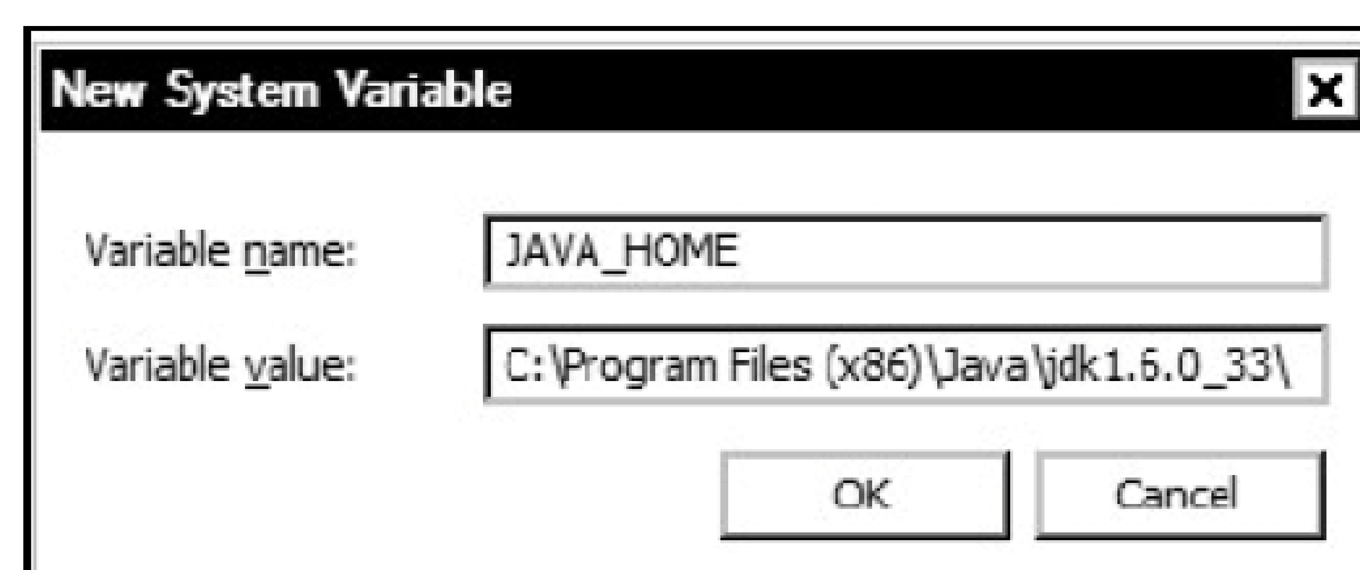


图 1-6　新的 JAVA_HOME 环境变量

(7) 单击 OK 按钮保存环境变量。

(8) 在系统变量列表中，双击 PATH 变量，并将;%JAVA_HOME%\bin 追加到变量值后面，如图 1-7 所示。

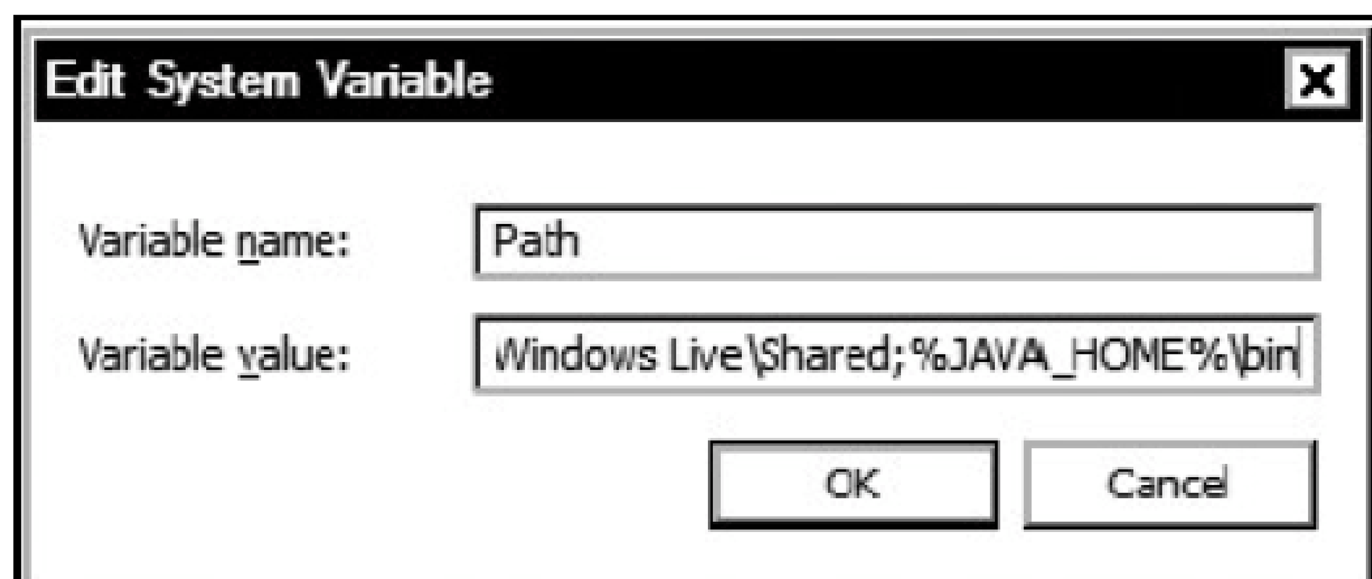


图 1-7　将 Oracle JDK 二进制路径追加到系统 PATH 变量中

现在 Oracle JDK 成为系统可执行文件搜索路径的一部分了，且该地址很容易找到。为了验证安装是否成功，选择 Start | Accessories | Command Prompt，打开一个命令提示窗口，在命令提示符下执行 javac –version。如果安装成功，就会看到 Oracle JDK 版本号，如图 1-8 所示。



图 1-8　Oracle JDK 安装的有效性验证

1.1.2　在 Windows 平台上下载并安装 Apache ANT

Apache ANT 是命令行构建工具，其任务是驱动根据目标和任务所描述的任何类型过程。Android 开发工具要求安装 Apache ANT 1.8 及以后版本，在本书编写时，最新版本是 Apache ANT 1.8.4。

请访问 http://ant.apache.org/bindownload.cgi 网站下载 Apache ANT，下载安装包为 ZIP 格式，如图 1-9 所示。安装步骤如下：



图 1-9　ZIP 格式的 Apache ANT 下载安装包

(1) Windows 操作系统支持 ZIP 文件，当下载完成时右击该 ZIP 文件。

(2) 在上下文菜单中选择 Extract All 打开 Extract Compressed Folder 向导。

(3) 单击 Browse 按钮，选择目标目录，如图 1-10 所示。因为 ZIP 文件已经包含一个名为 apache-ant-1.8.4 的目录用来保存 Apache ANT 文件，因此不需要建立专用的空白目标目录。本书中 C:\android 目录是用来保存 Android 开发工具及其相关工具的根目录，要记

住目标目录名以备后面设置环境变量时使用。

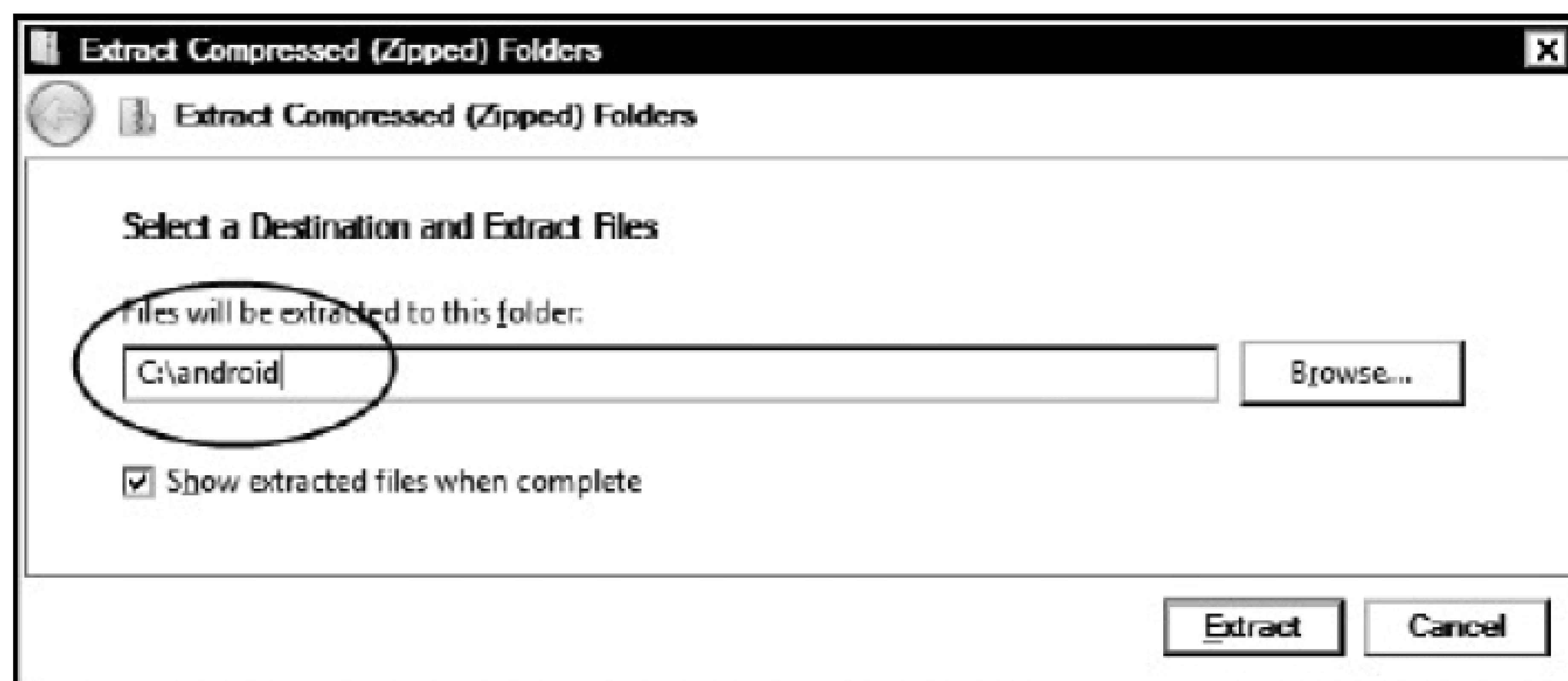


图 1-10　解压缩 Apache ANT ZIP 文档

(4) 单击 Extract 按钮安装 Apache ANT。

Apache ANT 安装完成后，按以下步骤将其二进制路径追加到系统可执行文件搜索路径中：

(1) 在 System Properties 界面打开 Environment Variables 对话框。

(2) 在系统变量部分单击 New 按钮定义一个新的环境变量。

(3) 将变量名设置成 ANT_HOME ，变量值设置成前面记下的 Apache ANT 安装目录(例如 C:\android\apache-ant-1.8.4)，如图 1-11 所示。

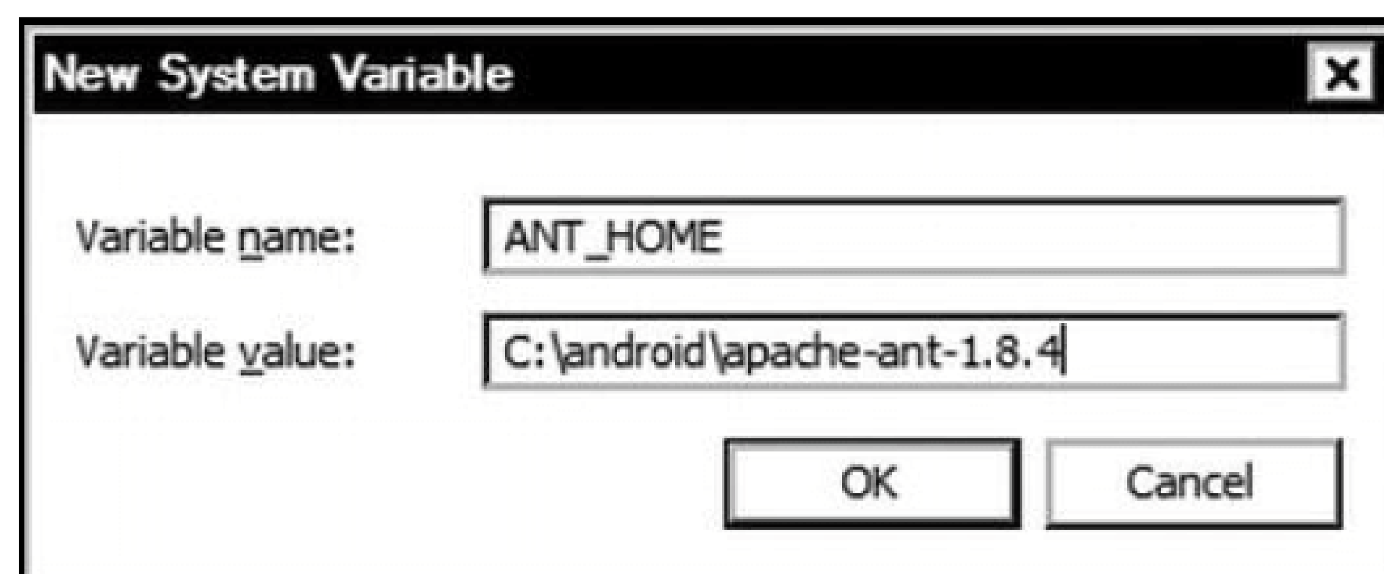


图 1-11　新建 ANT_HOME 环境变量

(4) 单击 OK 按钮保存新的环境变量。

(5) 在系统变量列表中，双击 PATH 变量，将;% ANT_HOME%\bin 追加到变量值后面，如图 1-12 所示。

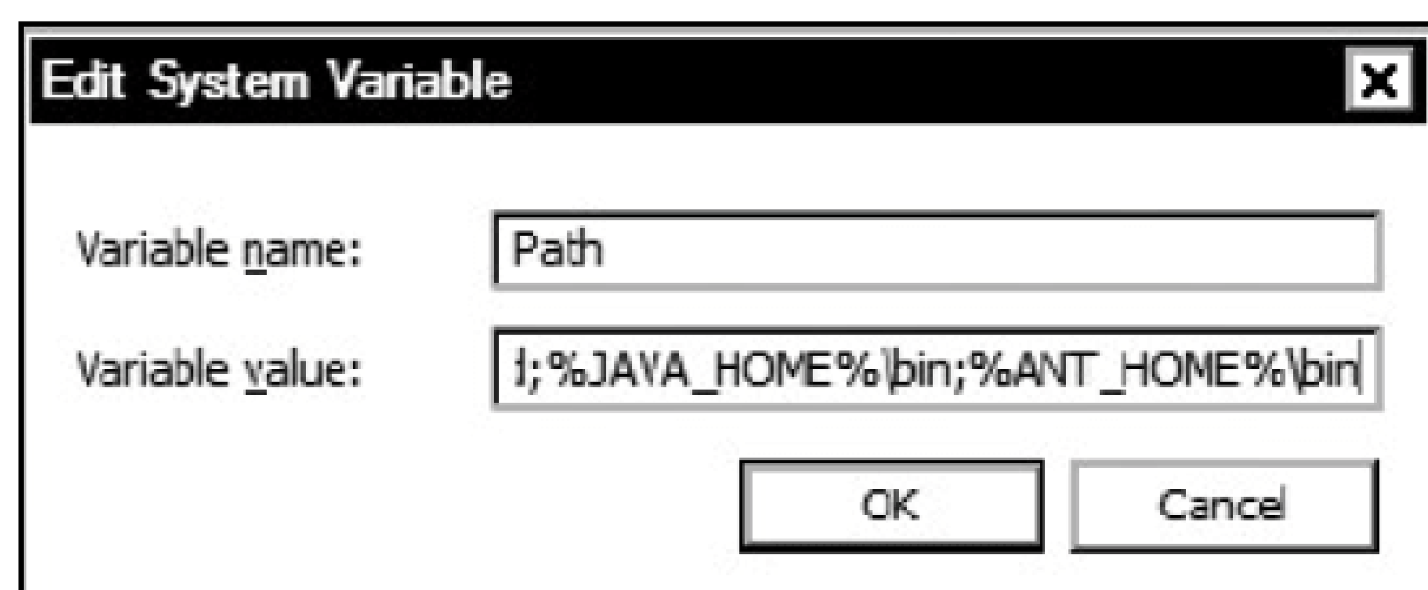


图 1-12　将 Apache ANT 二进制路径追加到系统 PATH 变量中

安装完成后，Apache ANT 被添加到系统可执行文件搜索路径中。为了验证安装是否成功，打开一个命令提示窗口。在命令提示符下执行 ant –version。如果安装成功，就会看到 Apache ANT 版本号，如图 1-13 所示。

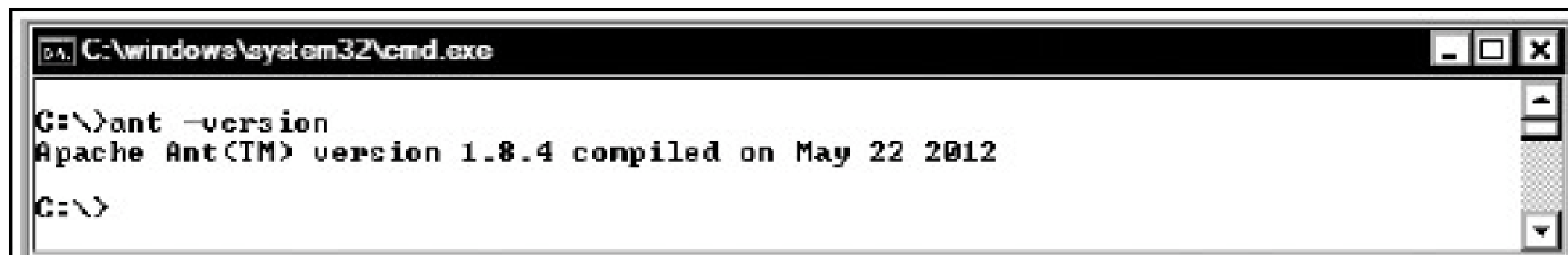


图 1-13　Apache ANT 安装的有效性验证

1.1.3　在 Windows 平台上下载并安装 Android SDK

Android 软件开发包(SDK)是开发工具链的核心组件，它提供框架 API 库和构建、测试及调试 Android 应用所需要的开发人员工具。

访问 http://developer.android.com/sdk/index.html 网站下载 Android SDK。本书编写时，Android SDK 的最新版本是 R20。当前提供两种类型的安装包：图形安装程序和 ZIP 文档。尽管图形安装程序被认为是主要的安装包，但是众所周知，在某些平台上它还存在问题。单击链接 Other platforms 并下载 Android SDK ZIP 文档，如图 1-14 所示。然后按照以下步骤进行操作：



图 1-14　Android SDK 下载页

(6) 下载完成后，右击 ZIP 文件，在上下文菜单中选择 Extract All 打开 Extract Compressed Folder 向导。

(7) 单击 Browse 按钮，选择目标目录。因为 ZIP 文件中已经包含一个名为 android-sdk-windows 的子目录，且其中包含 Android SDK 文件，因此不需要创建专用的空白目标目录。要记住目标目录名以备后面设置环境变量时使用。

(8) 单击 Extract 按钮安装 Android SDK。

安装完成后，按以下步骤将 Android SDK 的二进制路径追加到系统可执行文件搜索路径中。

(1) 在 System Properties 界面打开 Environment Variables 对话框。

(2) 在系统变量部分单击 New 按钮定义一个新的环境变量。

(3) 将变量名设置成 ANDROID_SDK_HOME，将变量值设置成前面记下的 Android SDK 安装目录(例如 C:\android\android-sdk-windows)，如图 1-15 所示。



图 1-15　ANDROID_SDK_HOME 环境变量

(4) 单击 OK 按钮保存新的环境变量。

(5) 有三个重要的目录需要添加到系统可执行搜索路径中：SDK根目录、保存Android平台独立SDK工具的工具目录和保存Android的平台工具目录，不考虑不存在平台工具目录的情况。在Environment Variables对话框中的系统变量列表中，双击PATH变量并将;%ANDROID_SDK_HOME%; %ANDROID_SDK_HOME%\tools; %ANDROID_SDK_HOME%\platform-tools追加到变量值后面，如图 1-16 所示。

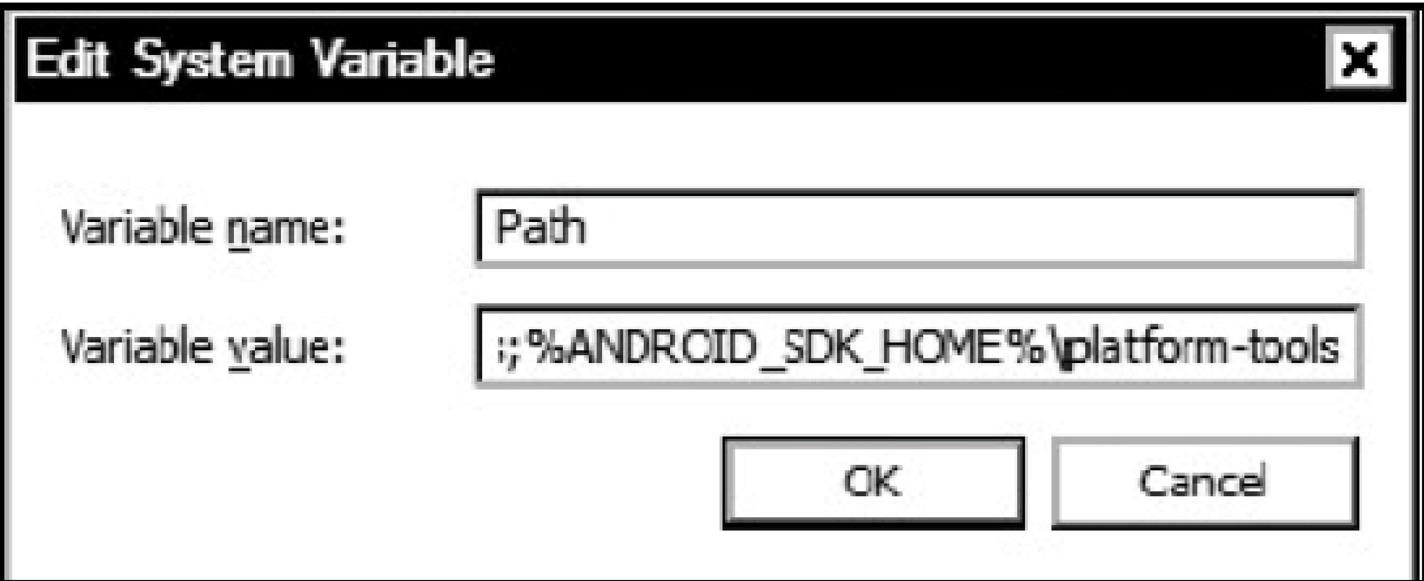


图 1-16　将 Android SDK 二进制路径追加到系统 PATH 变量中

为了验证安装是否成功，打开一个命令提示窗口，在命令提示符下执行'SDK Manager'(命令中包括引号)。如果安装成功，就会看到 Android SDK Manager 管理器，如图 1-17 所示。

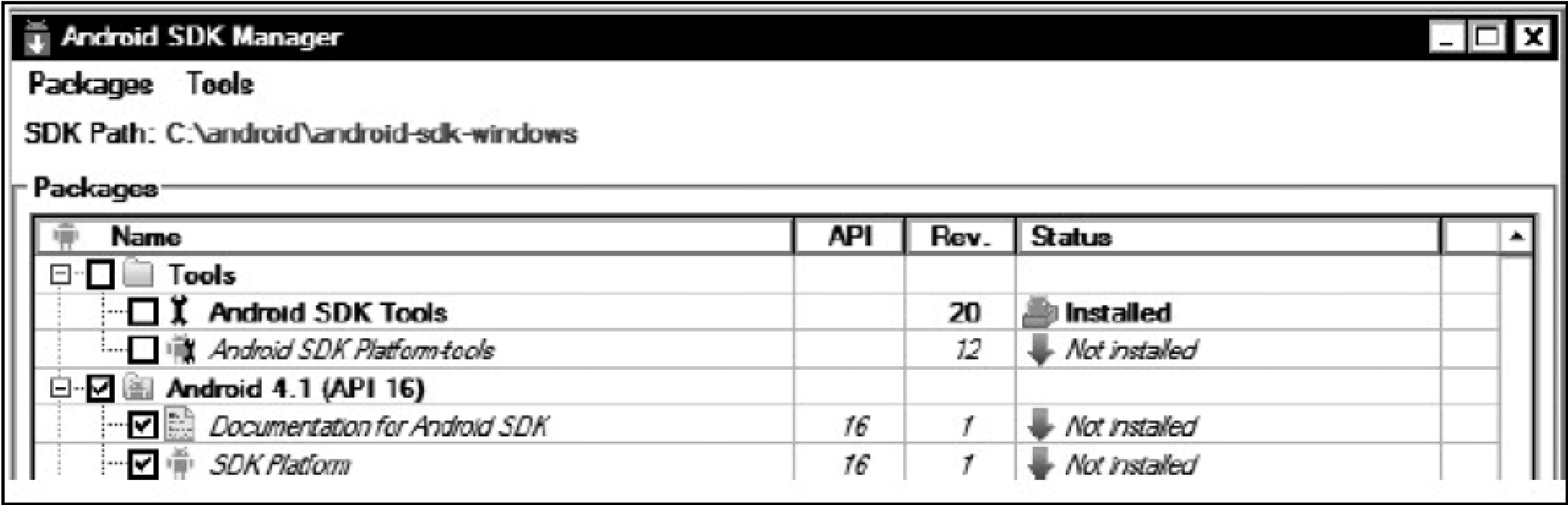


图 1-17　Android SDK Manager 应用

1.1.4　在 Windows 平台上下载并安装 Cygwin

Android原生开发包工具(Android Native Development Kit，NDK)最初设计在类UNIX系

统上工作，NDK的一些组件是shell脚本，这些脚本不能直接在Windows操作系统上执行。尽管Android NDK的最新版本表明它在独立性和自我打包方面有进步，但是它仍然需要在主机上安装Cygwin才能进行完整的操作。Cygwin是一个Windows操作系统上的类UNIX环境和命令行接口，它是基于UNIX应用程序的，包括允许运行Android NDK构建系统的shell。在本书编写时，Android NDK要求安装Cygwin 1.7 才能运行。访问 http://cygwin.com/install.html 网站并下载Cygwin安装程序setup.exe(如图 1-18 所示)。



图 1-18　下载 Cygwin 安装程序

启动安装程序之后，可以看见 Cygwin 安装向导欢迎界面，单击 Next 按钮按照以下步骤完成安装操作：

(1) 安装程序会让用户选择下载源，选择默认选项 Install from Internet，并单击 Next 按钮继续。

(2) 在下一个对话框中，安装程序会让用户选择 Cygwin 的安装目录，如图 1-19 所示。默认情况下，Cygwin 被安装在 C:\cygwin 目录下。注意要记住目标目录名以备后面使用，然后单击 Next 按钮。

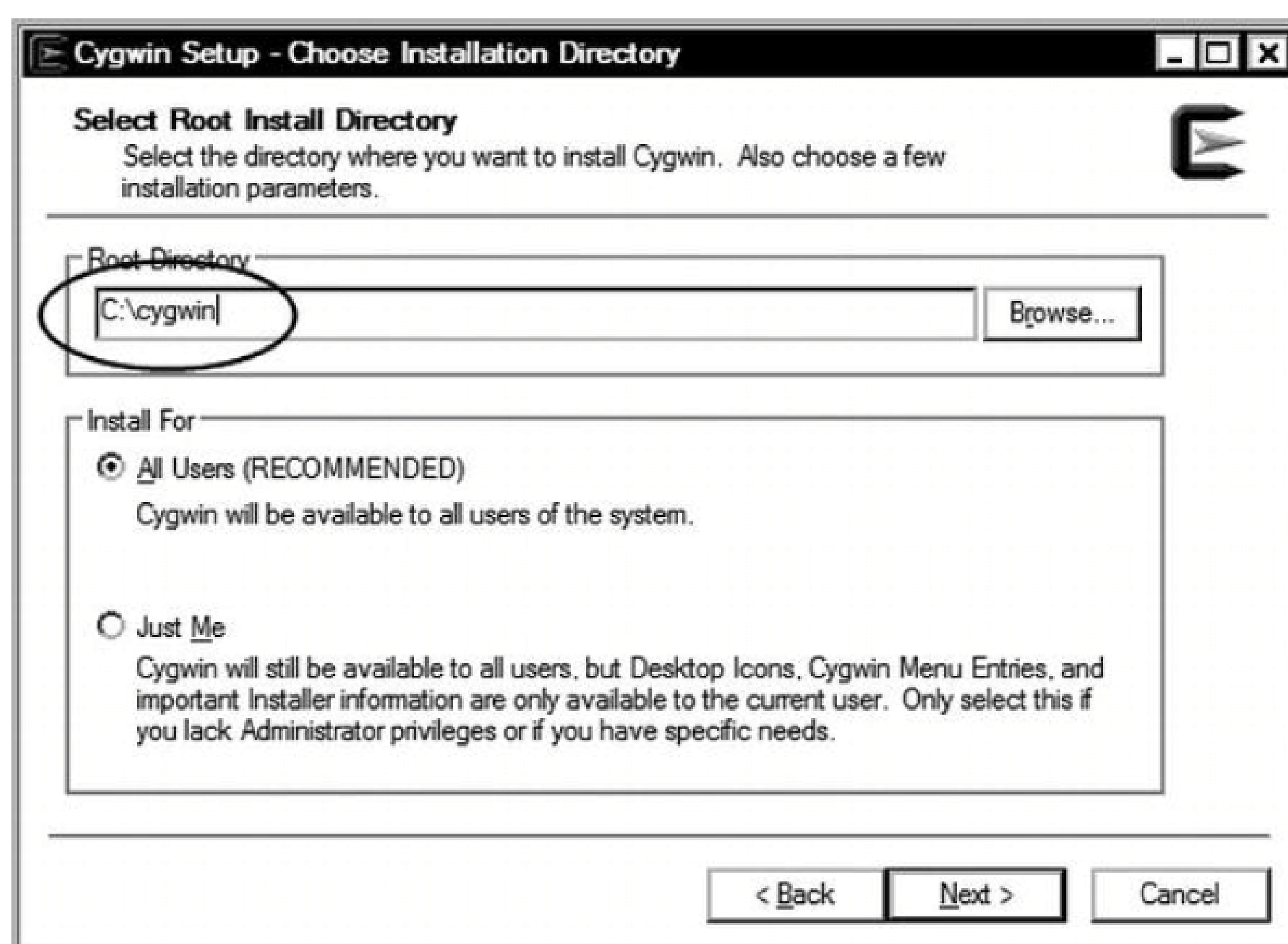


图 1-19　选择 Cygwin 安装目录

(3) 下一个对话框将让用户选择原生包目录，这是一个用于下载文件包的临时目录。使用默认值，然后单击 Next 按钮。

(4) 下一个对话框将让用户选择 Internet 连接类型，除非需要用代理访问 Internet，否则选择默认项 Direct Connection，然后单击 Next 按钮继续。

(5) 安装程序将让用户选择下载站点，从镜像站点列表中随机选择一个站点或者选择一个离安装站点地理位置最近的站点，然后单击 Next 按钮。

(6) Cygwin 不是单个的应用程序，是包含多个应用程序的巨大的软件分布。在下一个对话框中，Cygwin 安装程序会为用户提供一个可用包列表，Android NDK 要求安装 GNU Make 3.8.1 及以后版本以正常使用其功能。在搜索框中输入关键字 make 对包列表进行过滤，展开 Devel 目录，选择 GNU Make 包，如图 1-20 所示。单击 Next 按钮开始安装。

安装完成后，要把 Cygwin 二进制路径添加到系统可执行搜索路径中。

(1) 从 System Properties 打开 Environment Variables 对话框。

(2) 在系统变量部分单击 New 按钮定义一个新的环境变量。

(3) 将变量名设置成 CYGWIN_HOME，将变量值设置成 Cygwin 安装目录(例如 C:\cygwin)，如图 1-21 所示。

(4) 在 Environment Variables 对话框中的系统变量列表中双击 PATH 变量，并将 ;%CYGWIN_HOME%\bin 追加到变量值后面，如图 1-22 所示。

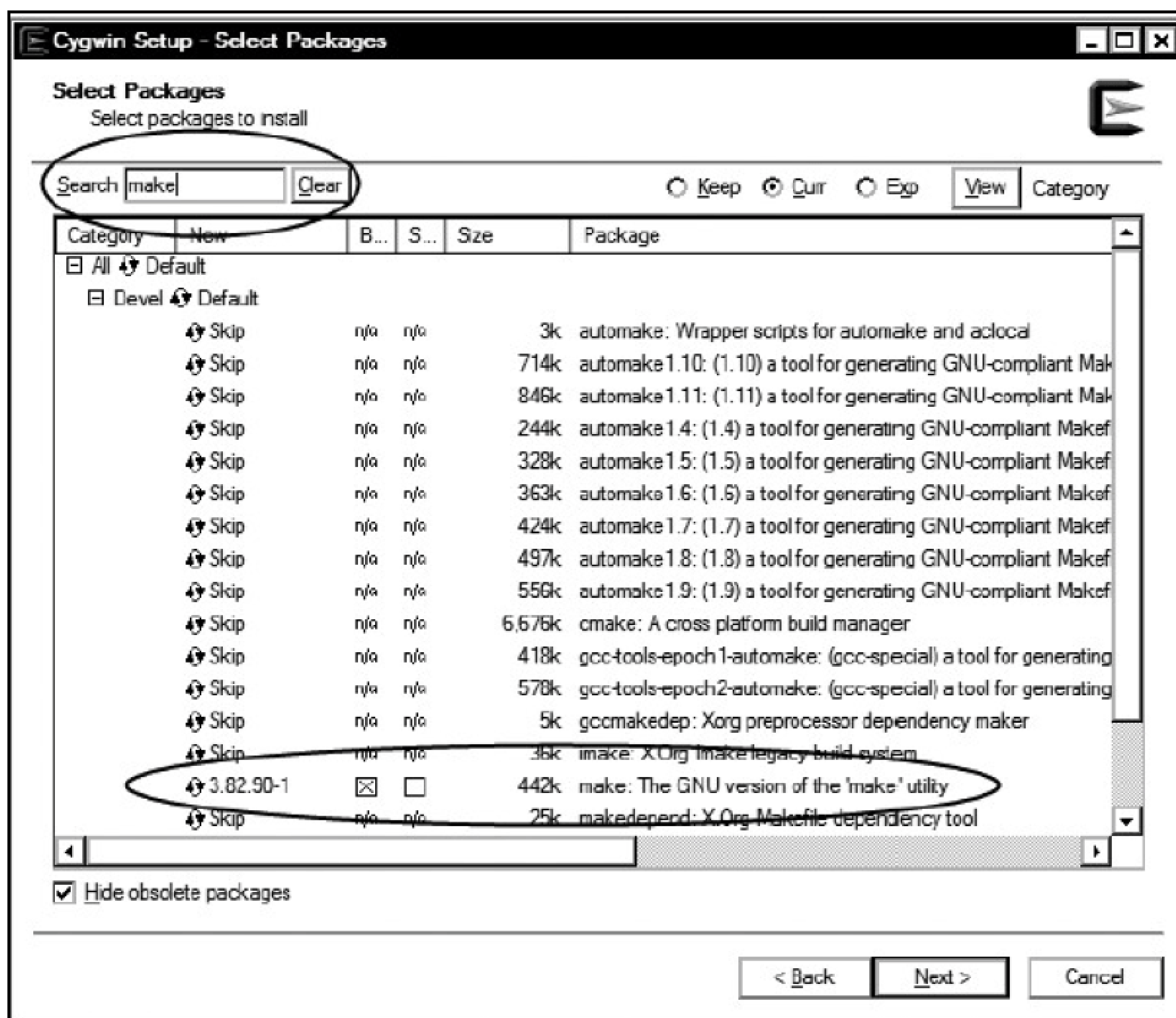


图 1-20　选择 GNU Make 包

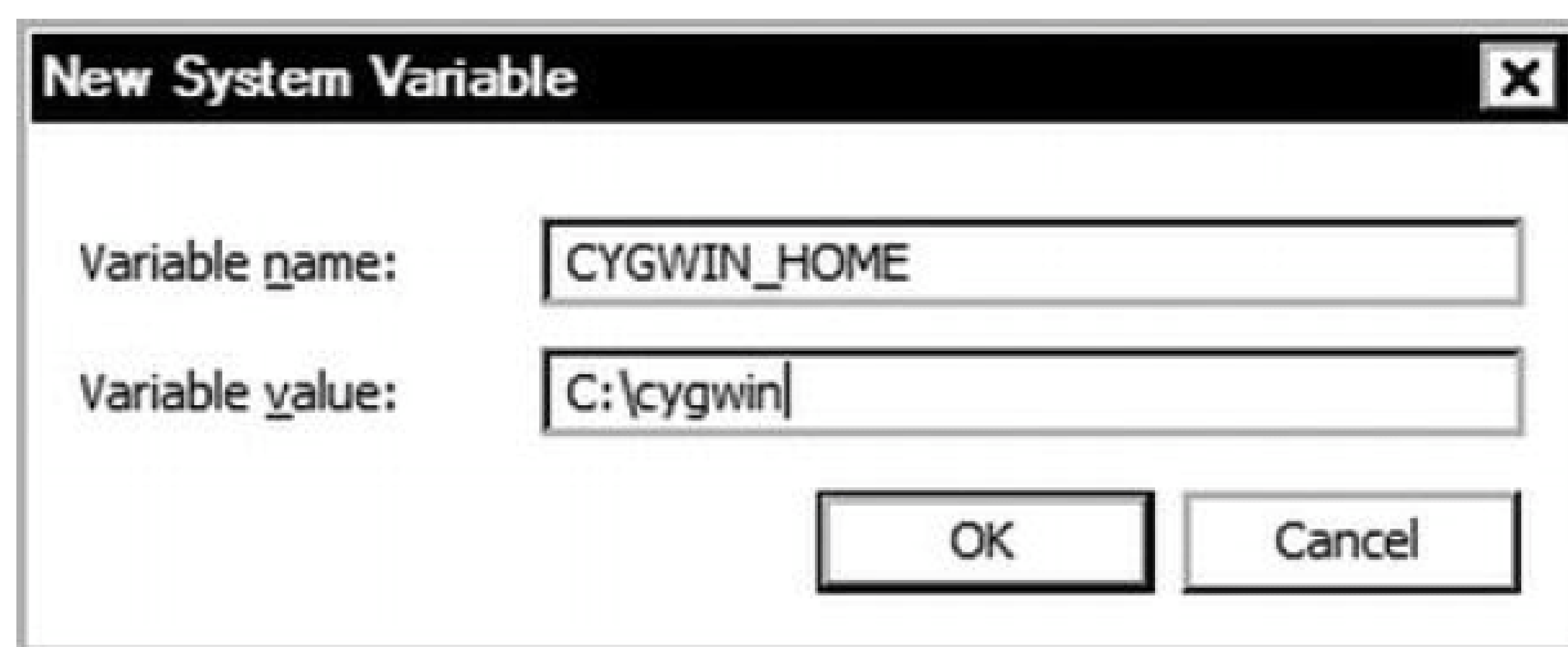


图 1-21　CYGWIN_HOME 环境变量

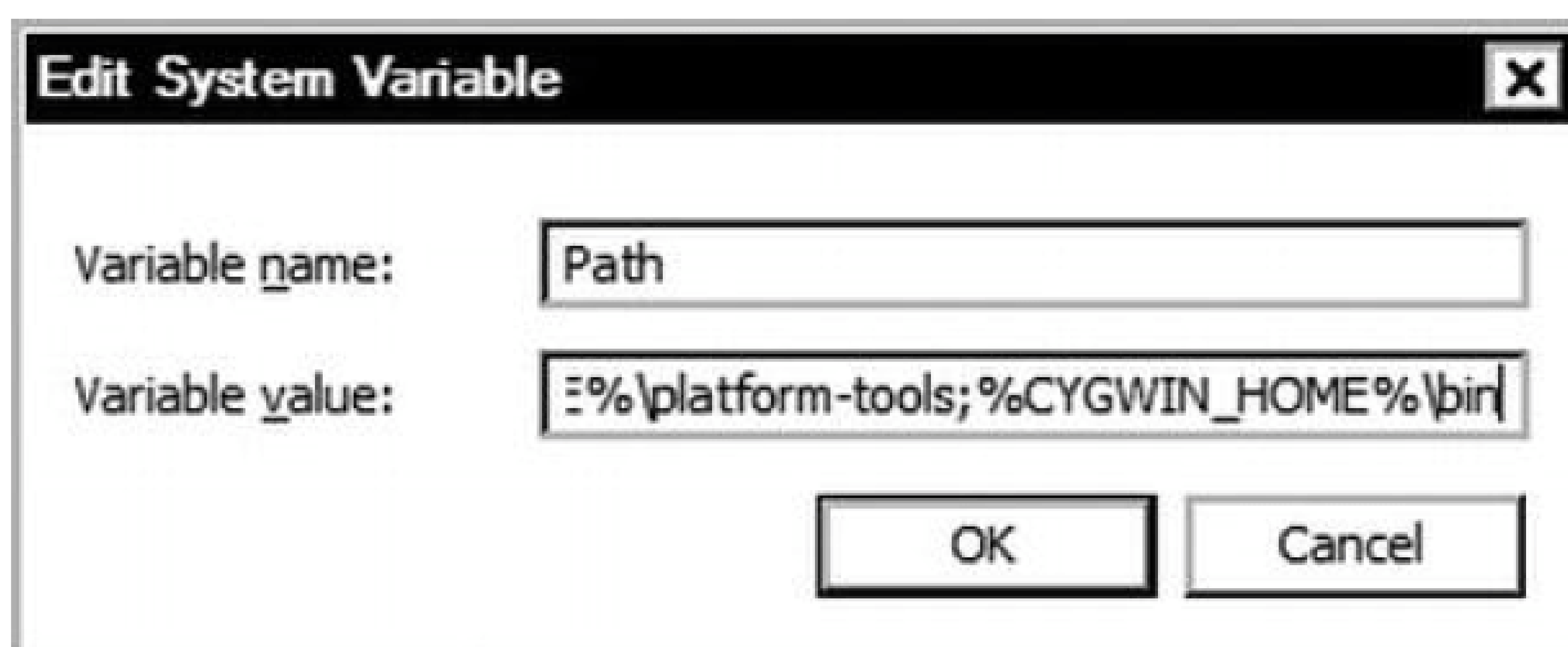


图 1-22　将 Cygwin 二进制路径追加到系统 PATH 变量中

完成了上述安装步骤后，Cygwin 工具成为系统可执行搜索路径的一部分。为了验证安装是否成功，打开一个命令提示窗口，在命令提示符下执行 make –version。如果安装成功，则会显示 GNU Make 的版本号，如图 1-23 所示。

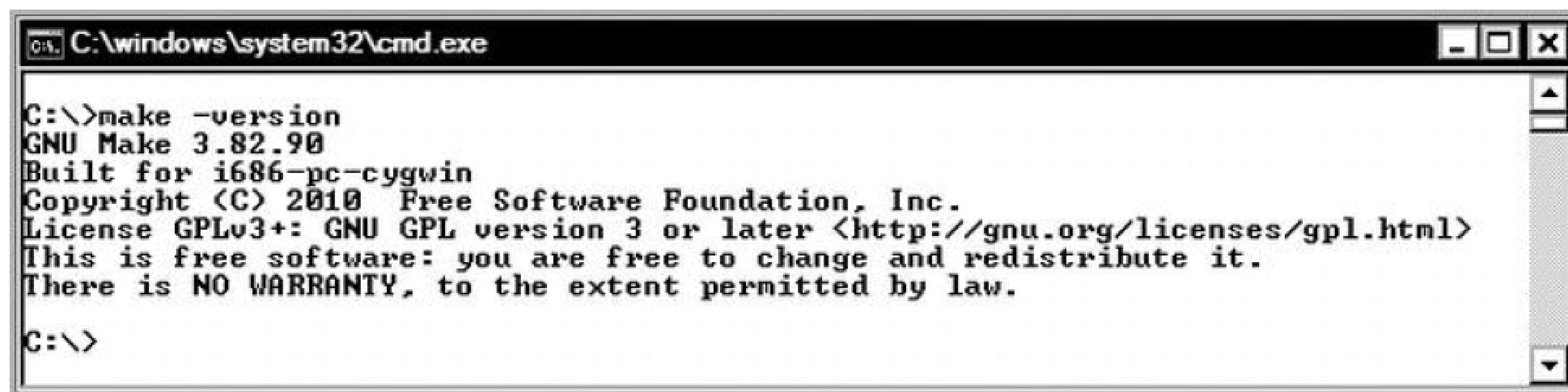


图 1-23　验证 Cygwin 安装结果

1.1.5　在 Windows 平台上下载并安装 Android NDK

Android 原生开发工具包(Native Development Kit，NDK)是 Android SDK(Software Development Kit)的伴随工具，它可以让用户用诸如 C++的原生编程语言开发 Android 应用程序。Android NDK 提供了头文件、库和交叉编译器工具链。本书编写时，Android NDK 的最新版本是 R8。请定位到 http://developer.android.com/tools/sdk/ndk/index.html 网站并找到图 1-24 所示的 Downloads 部分。下载步骤如下：



图 1-24　Android NDK 下载页面

(1) Android NDK 安装包以 ZIP 文档的形式提供。下载完成时，右击 ZIP 文件并在上下文菜单中选择 Extract All 选项，打开 Extract Compressed Folder 向导。

(2) 用 Browse 按钮选择目标目录。因为 ZIP 文件中已经包含一个名为 android-ndk-r8 的子目录，其中包含 Android NDK 文件，因此不需要建立专用的空白目标目录。要记住目标目录名以备后面设置环境变量时使用。

(3) 单击 Extract 按钮安装 Android NDK。

安装完成后，按以下步骤将 Android SDK 的二进制路径追加到系统可执行搜索路径中：

(1) 同样，在 System Properties 界面打开 Environment Variables 对话框。

(2) 在系统变量部分单击 New 按钮定义一个新的环境变量，将变量名设置成 ANDROID_NDK_HOME，将变量值设置成前面记下的 Android NDK 安装目录(例如 C:\android\android-ndk-r8)，如图 1-25 所示。

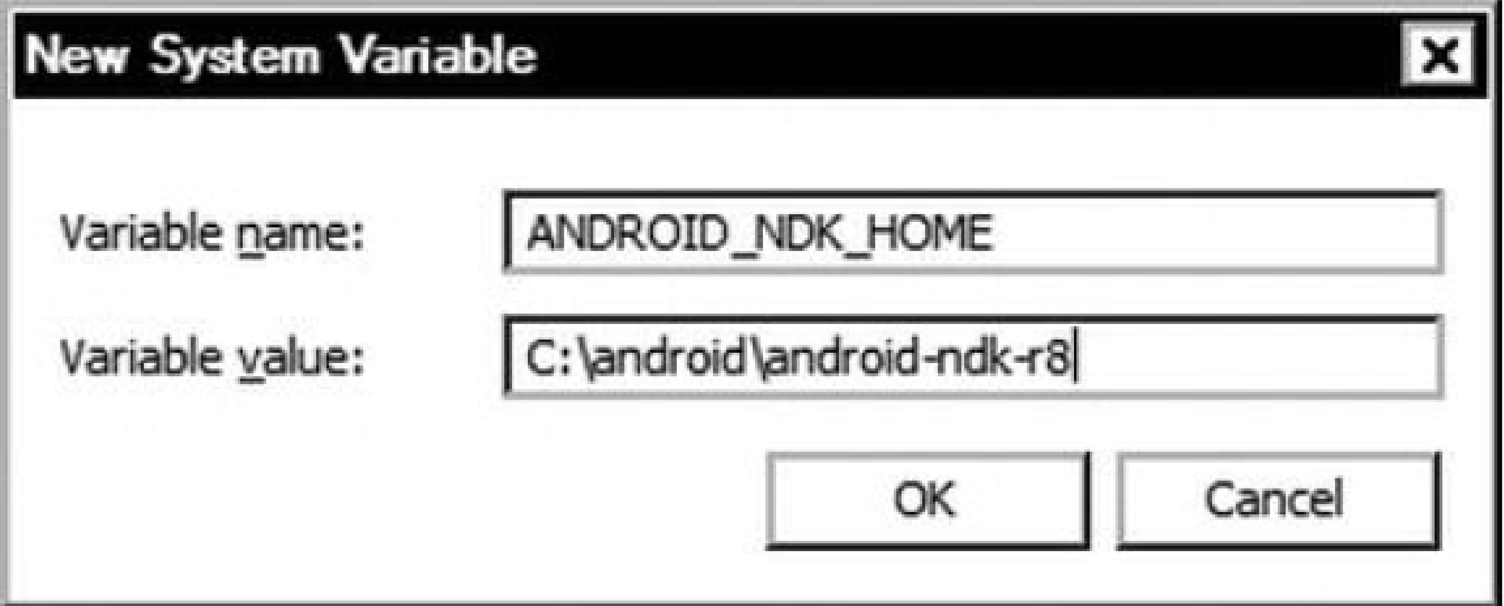


图 1-25　ANDROID_NDK_HOME 环境变量

(3) 单击 OK 按钮保存该新环境变量。

(4) 在环境变量对话框中的系统变量列表中，双击 PATH 变量，并将;%ANDROID_NDK_HOME%追加到变量值后面，如图 1-26 所示。

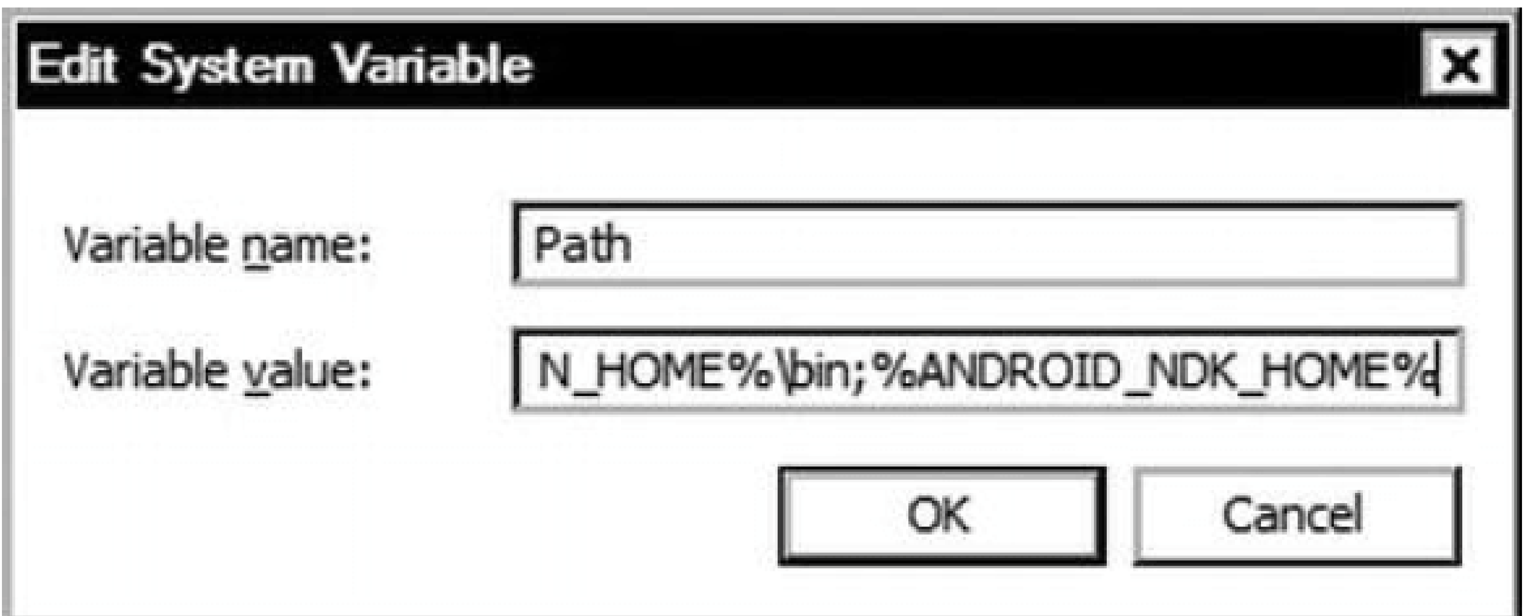


图 1-26　将 Android NDK 二进制路径追加到系统 PATH 变量中

现在可以很容易地访问 Android NDK。为了验证安装是否成功，打开一个命令提示窗口，在命令提示符下执行 ndk-build。如果安装成功，就会看到 NDK 给出的关于项目目录的提示，如图 1-27 所示。

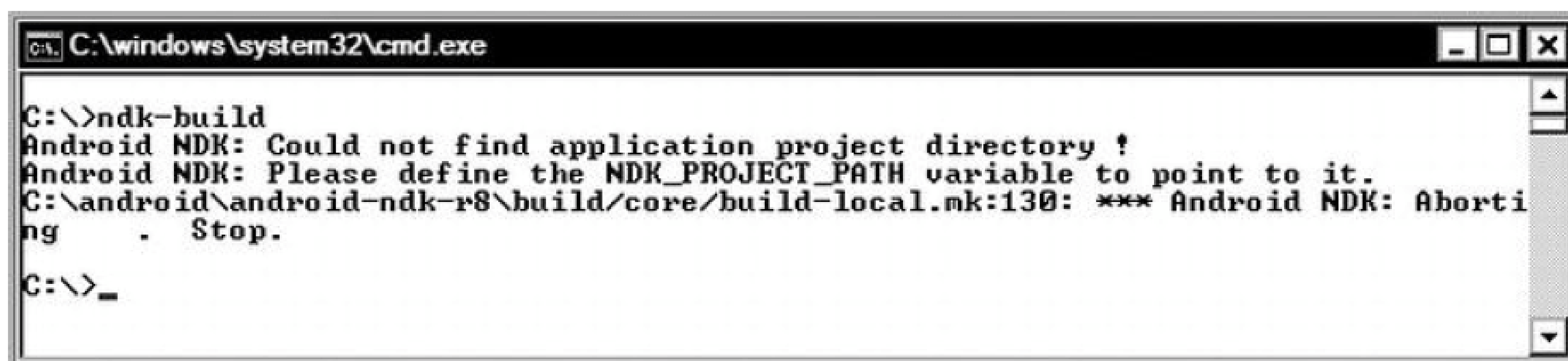


图 1-27　Android NDK 安装结果验证

1.1.6　在 Windows 平台上下载并安装 Eclipse

Eclipse 是高度可扩展的、多语言集成的开发环境，尽管原生 Android 开发不要求必须安装 Eclipse，但是因为 Eclipse 提供了高度集成的编码环境，将其与 Android 工具结合使用可以简化应用程序的开发过程。本书编写时，Eclipse 的最新版本是 Juno 4.2，请访问 http://www.eclipse.org/downloads/网站下载 Eclipse，如图 1-28 所示，下载步骤如下：

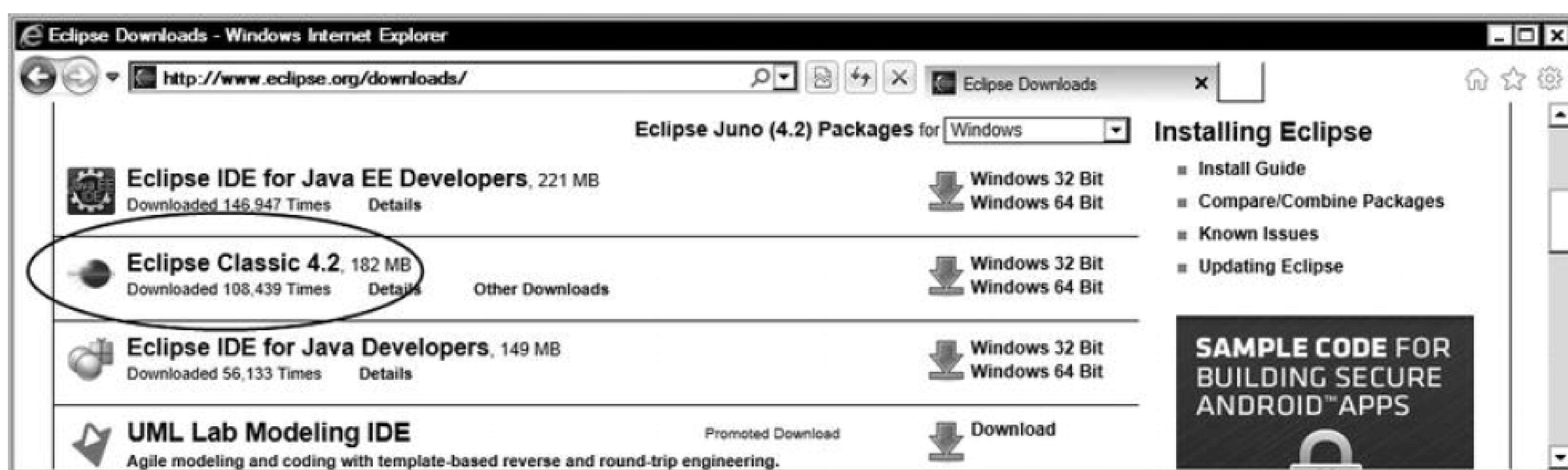


图 1-28　Eclipse 下载页面

(1) 从列表中下载 Eclipse Classic for Windows 32 Bit，Eclipse 安装包以 ZIP 文档形式提供。

(2) 下载完成时，右击该 ZIP 文件，并在上下文菜单中选择 Extract All 选项，打开 Extract Compressed Folder 向导。

(3) 用 Browse 按钮选择目标目录，因为 ZIP 文件中已经包含一个名为 eclipse 的子目录，其中包含 Eclipse 文件，因此不需要建立专用的空白目标目录。要记住目标目录名以备后面设置环境变量时使用。

(4) 单击 Extract 按钮安装 Eclipse。

(5) 为了方便访问 Eclipse，进入 Eclipse 安装目录。

(6) 右击 Eclipse 二进制，并选择 Send | Desktop 在 Windows 桌面上建立 Eclipse 的快捷键。

为了验证 Eclipse 安装结果的有效性，双击 Eclipse 图标。如果安装成功，将看到图 1-29

所示的 Eclipse Workspace Launcher 对话框。

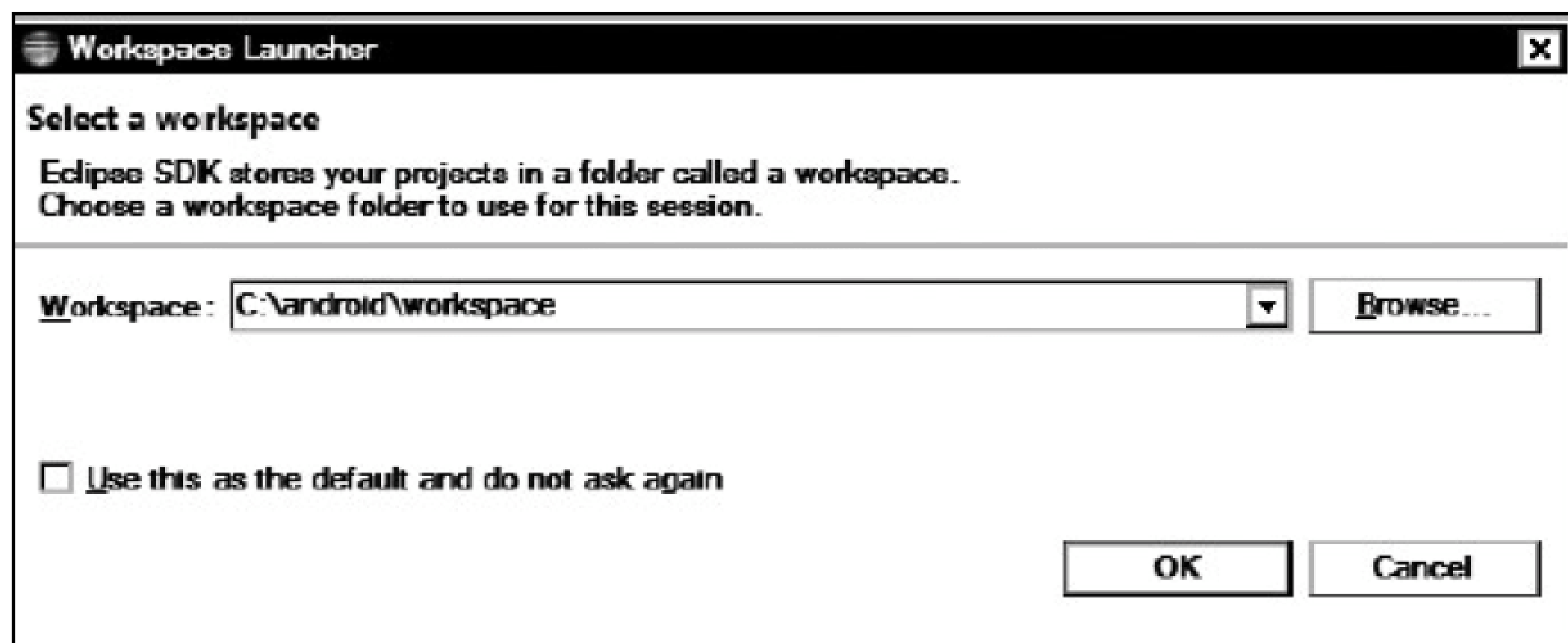


图 1-29　Eclipse 安装结果验证

1.2　Apple Mac OS X

Android 开发工具要求 Mac OS X 10.5.8 及后续版本和 x86 系统。因为 Android 开发工具最初被设计成在类 UNIX 操作系统上工作，可以直接通过 OS X 或者 Xcode 开发工具在平台上使用它的大多数扩展功能。在本节中，用户需要下载并安装以下组件：

- Xcode
- Java JDK 6
- Apache ANT Build System
- GNU Make
- Android SDK
- Android NDK
- Eclipse IDE

1.2.1　在 Mac 平台上安装 Xcode

Xcode 可以为 OS X 平台上的应用程序开发提供开发工具。它可以在 Mac OS X 安装介质中找到，或者通过 Mac App Store 免费获取。访问 https://developer.apple.com/xcode/网站可以获取更多信息。启动 Xcode 安装程序将进入 Xcode 安装向导，向导程序将引导你完成安装过程。

(1) 同意许可协议。

(2) 选择目标目录。

(3) 安装向导会显示可以安装的 Xcode 组件列表，从列表中选择 UNIX 开发工具包，如图 1-30 所示。

(4) 单击 Continue 按钮开始安装。

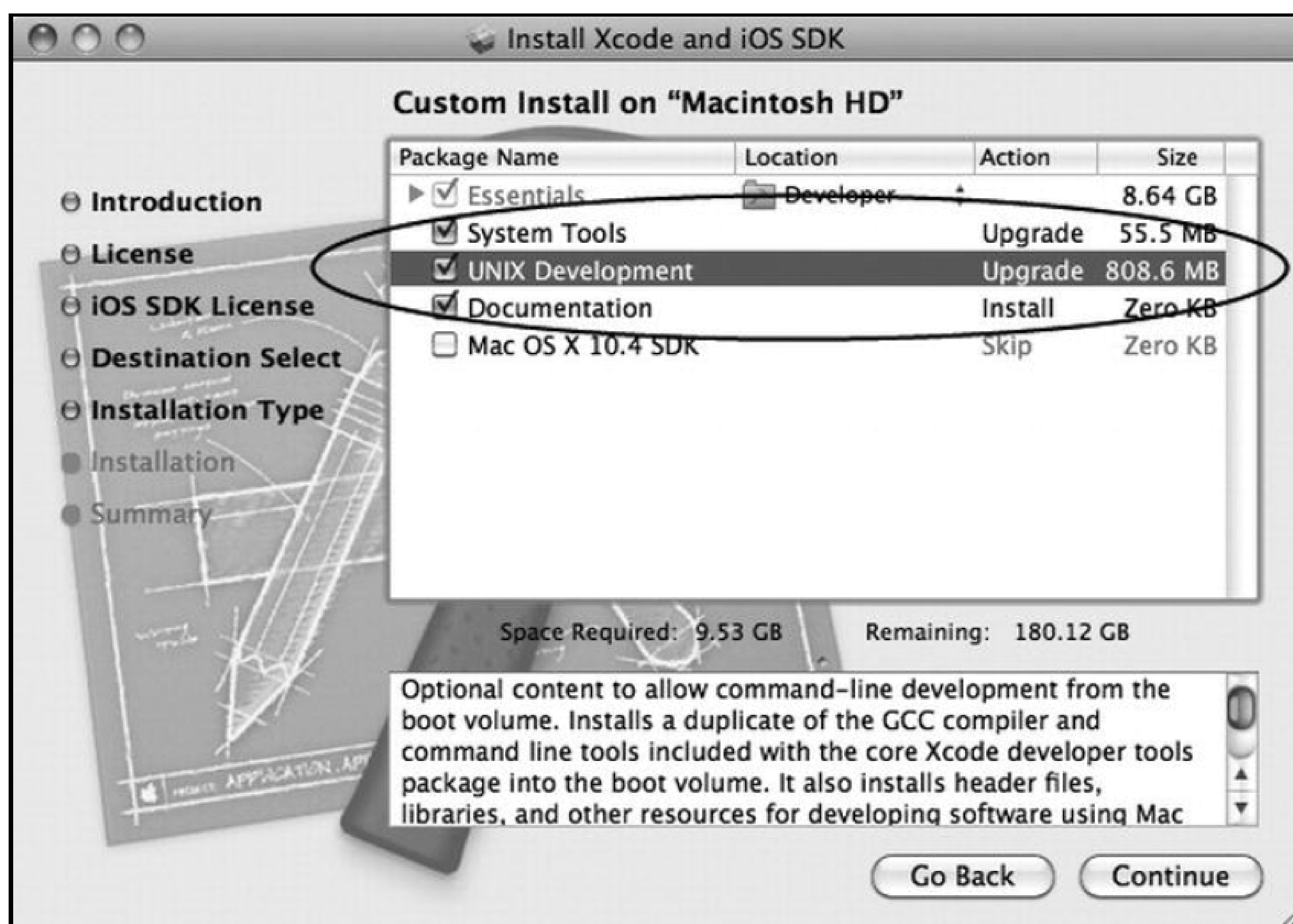


图 1-30　Xcode 定制安装程序对话框

1.2.2　验证 Mac 平台的 Java 开发包

Android 开发工具要求事先安装 JDK(Java Development Kit)6 才能运行，苹果 Mac OS X 操作系统已经配置了 JDK，所配置的 JDK 是基于 Oracle JDK 的，但是由苹果来配置 JDK 可以更好地与 Mac OS X 集成。可以通过 Software Update 获取 JDK 的新版本。必须确保安装的是 JDK 6 及以后版本。为了验证 JDK 的安装效果，打开一个终端窗口，在命令行方式下执行 javac –version。如果 JDK 安装正确，会看到 JDK 的版本号，如图 1-31 所示。

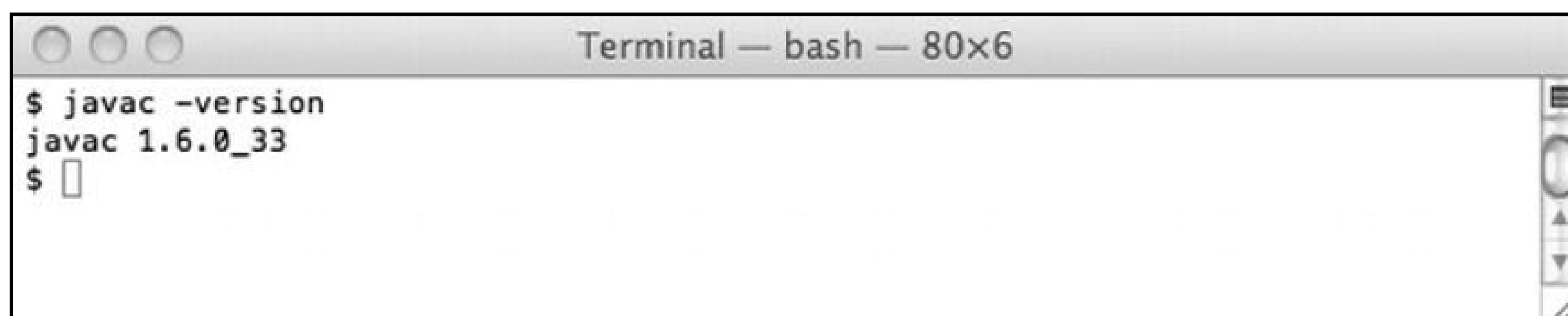


图 1-31　验证 JDK

1.2.3　验证 Mac 平台上的 Apache ANT

Apache ANT 是命令行构建工具，它可以驱动任何根据目标和任务描述的过程，Android 开发工具要求安装 Apache ANT 1.8 及以后版本才能运行构建过程，Apache ANT 作为 Xcode 的 UNIX 开发包的一部分安装到系统中。为了验证 Apache ANT 的安装效果，打开一个终端窗口并在命令行方式下执行 ant –version。如果安装成功，会看到 Apache ANT 的版本号，如图 1-32 所示。

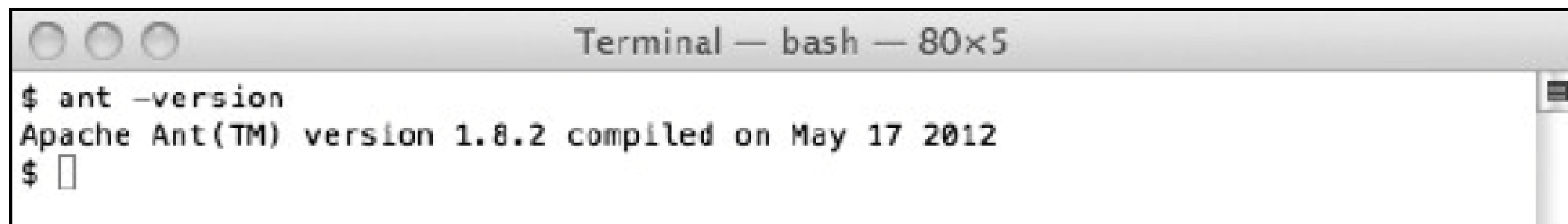


图 1-32　验证 Apache ANT

1.2.4　验证 GNU Make

GNU Make 是在应用程序源代码中控制生成程序中的可执行程序部分及其他部分的构建工具。Android NDK 要求安装 GNU Make 3.8.1 及以后版本。GNU Make 是作为 Xcode 的 UNIX 开发包的一部分安装到系统中。为了验证 GNU Make 的安装效果，打开一个终端窗口，在命令行方式下执行 make –version。如果安装成功，会看到 GNU Make 的版本号，如图 1-33 所示。

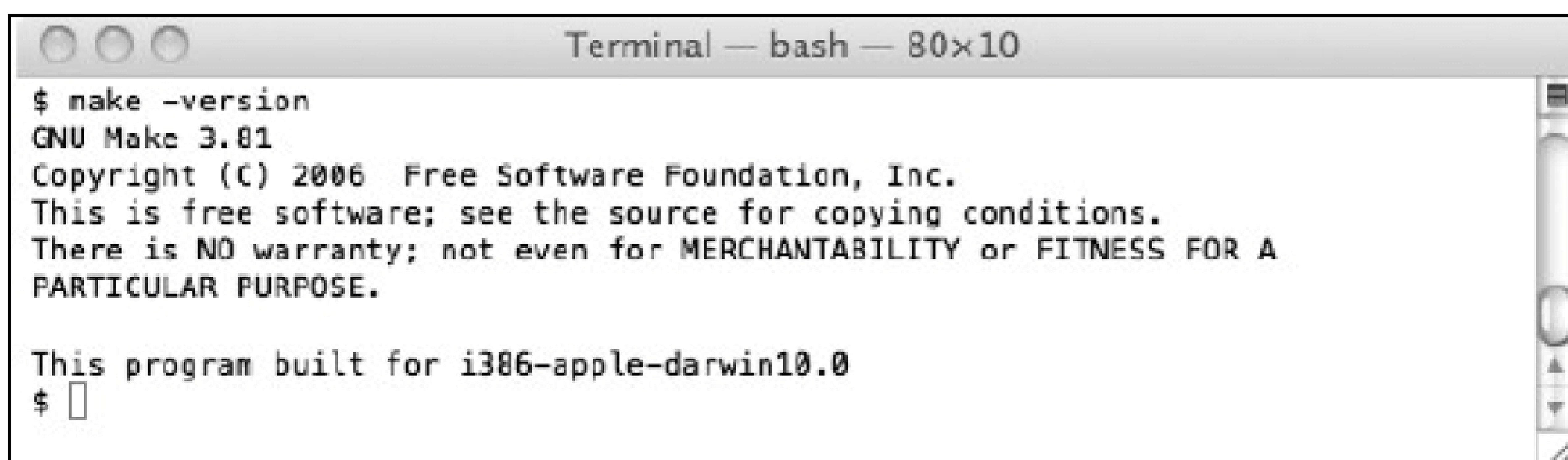


图 1-33　验证 GNU Make

1.2.5　在 Mac 平台上下载并安装 Android SDK

Android 软件开发包(SDK)是开发工具链的核心组件，它提供了构建、测试和调试 Android 应用程序所需要的框架 API 库和开发工具。本书编写时，Android SDK 的最新版本是 R20，请访问 http://developer.android.com/sdk/index.html 网站下载 Android SDK，如图 1-34 所示。安装步骤如下：



图 1-34　Android SDK 下载页面

(1) 单击 Download the SDK for Mac 按钮开始下载 SDK 安装包。

(2) Android SDK 安装包以 ZIP 文档方式提供，OS X 提供对 ZIP 文档的原生支持。如果使用 Safari 浏览器，则 ZIP 文件会在下载后自动解压，也可以双击下载的 ZIP 文件以压缩文件夹的形式打开文件。

(3) 使用 Finder 将 android-sdk-macosx 目录拖放到目标文件夹下，如图 1-35 所示。本书中/android 目录是保存 Android 开发工具及其相关组件的根目录。

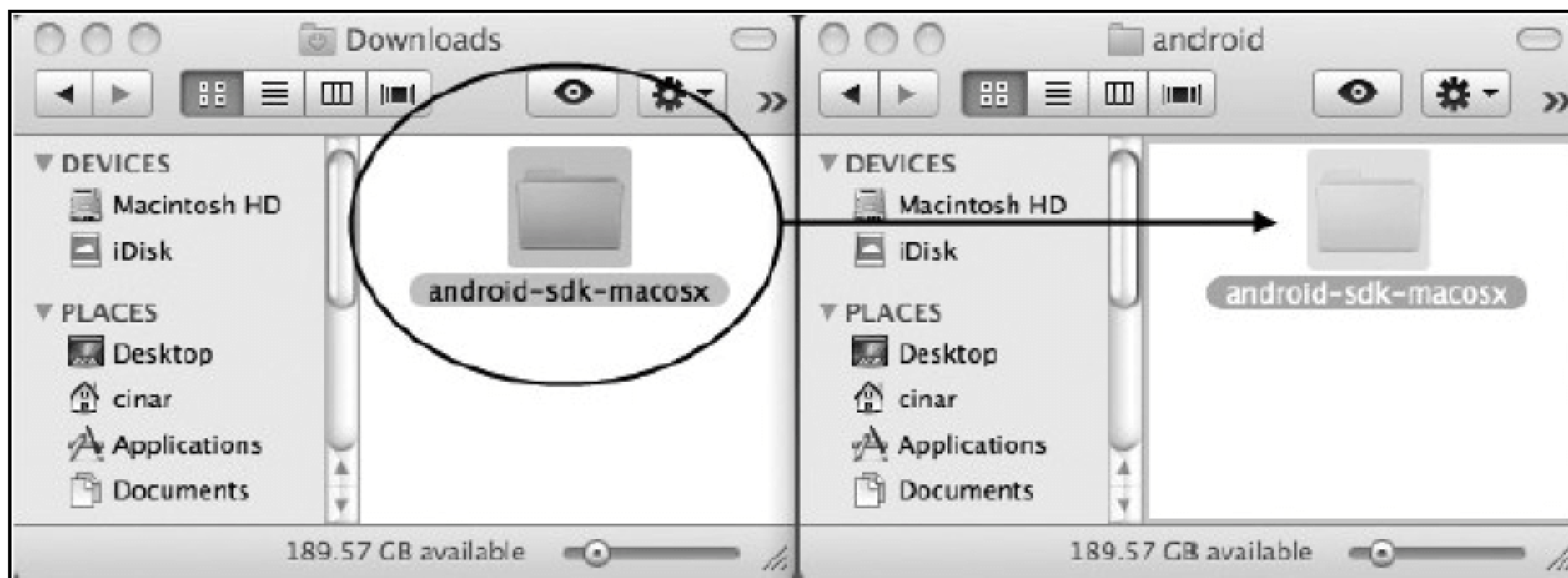


图 1-35　将 Android SDK 安装到目标位置

为了便于访问 Android SDK，要把 Android SDK 二进制路径追加到系统可执行搜索路径中。打开一个终端窗口并执行下列命令，如图 1-36 所示：

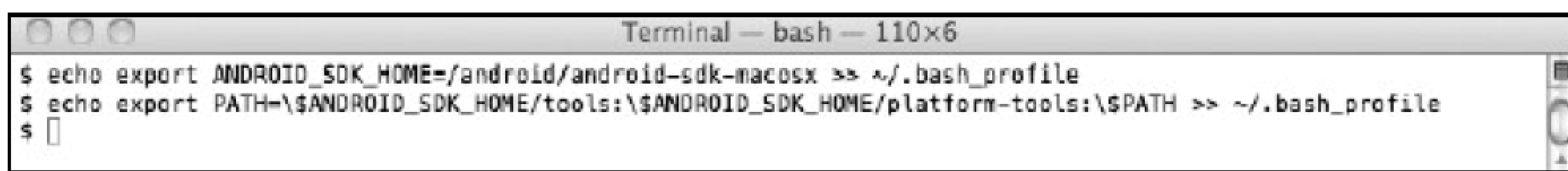


图 1-36　将 SDK 二进制路径追加到系统 PATH 变量中

- echo export ANDROID_SDK_HOME=/android/android-sdk-macosx > > ~/.bash_profile
- echo export PATH = \$ANDROID_SDK_HOME/tools:\$ANDROID_SDK_HOME/platformtools:\$PATH > > ~/.bash_profile

为了验证 Android SDK 的安装效果，打开一个新的终端窗口并在命令行方式下执行 android –h。如果安装成功，将会看到图 1-37 显示的帮助信息。

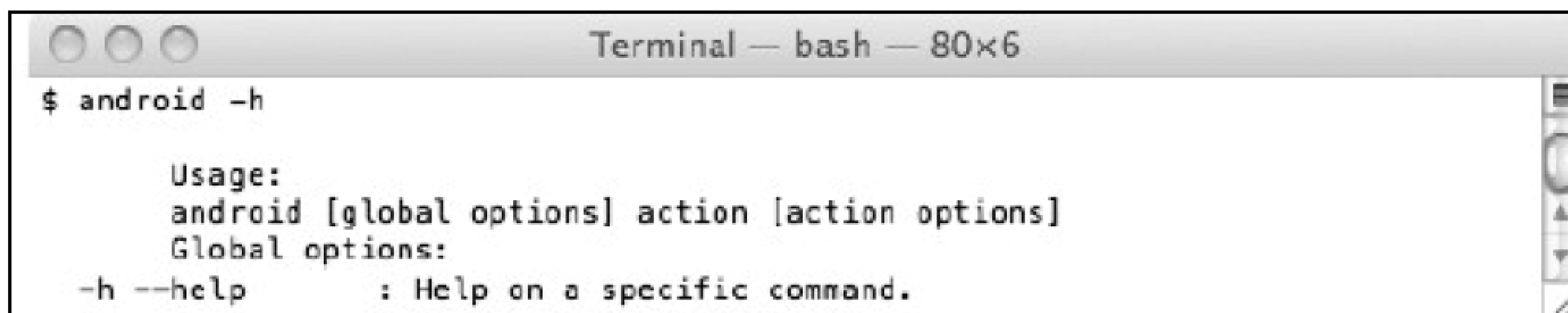


图 1-37　验证 Android SDK 安装效果

1.2.6 在 Mac 平台上下载并安装 Android NDK

Android 原生开发工具包(NDK)是 Android SDK 的伴随工具，它可以让用户用诸如 C++之类的原生编程语言开发 Android 应用程序。Android NDK 提供头文件、库和交叉编译器工具链。本书编写时，Android NDK 的最新版本是 R8。请到 http://developer.android.com/tools/sdk/ndk/index.html 网站下载 Android NDK，下载部分如图 1-38 所示。下载步骤如下：



图 1-38　Android NDK 下载页面

(1) 单击下载安装包，Android NDK 安装包以 BZIP'ed TAR 文档形式提供，OS X 不会自动解压这种类型的文件。

(2) 为了自动解压文件，打开一个终端窗口。

(3) 进入目标目录/android。

(4) 执行 tar jxvf ~/Downloads/android-ndk-r8-darwin-x86.tar.bz2，如图 1-39 所示。



图 1-39　安装 Android NDK

为了便于访问，要把 Android NDK 二进制路径追加到系统可执行搜索路径中，打开终端窗口并执行下列命令(如图 1-40 所示)。

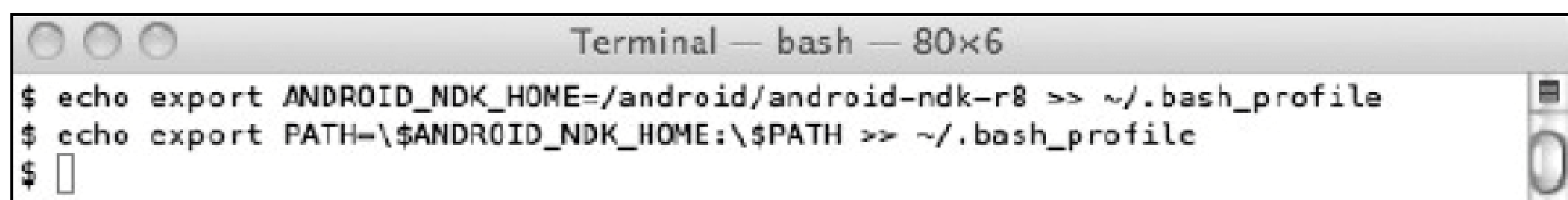


图 1-40　将 Android NDK 二进制路径追加到系统 PATH 变量中

- echo export ANDROID_NDK_HOME=/android/android-ndk-r8 > > ~/.bash_profile
- echo export PATH = \$ANDROID_NDK_HOME:\$PATH > > ~/.bash_profile

打开一个新的终端窗口并在命令行方式下执行命令 ndk-build 以验证 Android NDK 的安装效果。如果安装成功，就会看到 NDK 给出的关于项目目录的提示，如图 1-41 所示。

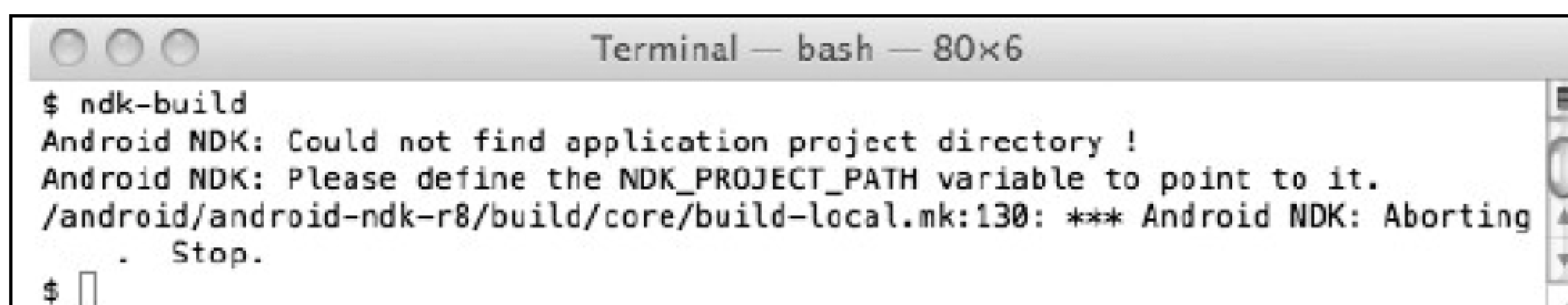


图 1-41　验证 Android NDK

1.2.7　在 Mac 平台上下载并安装 Eclipse

Eclipse 是高度可扩展的、多语言集成开发环境，尽管原生 Android 开发不要求必须安装 Eclipse， 但是 Eclipse 提供了高度集成的编码环境，它与 Android 工具结合使用可以简化应用程序的开发过程。本书编写时，Eclipse 的最新版本是 Juno 4.2。请访问 http://www.eclipse.org/downloads/网站下载 Eclipse，如图 1-42 所示，下载步骤如下：

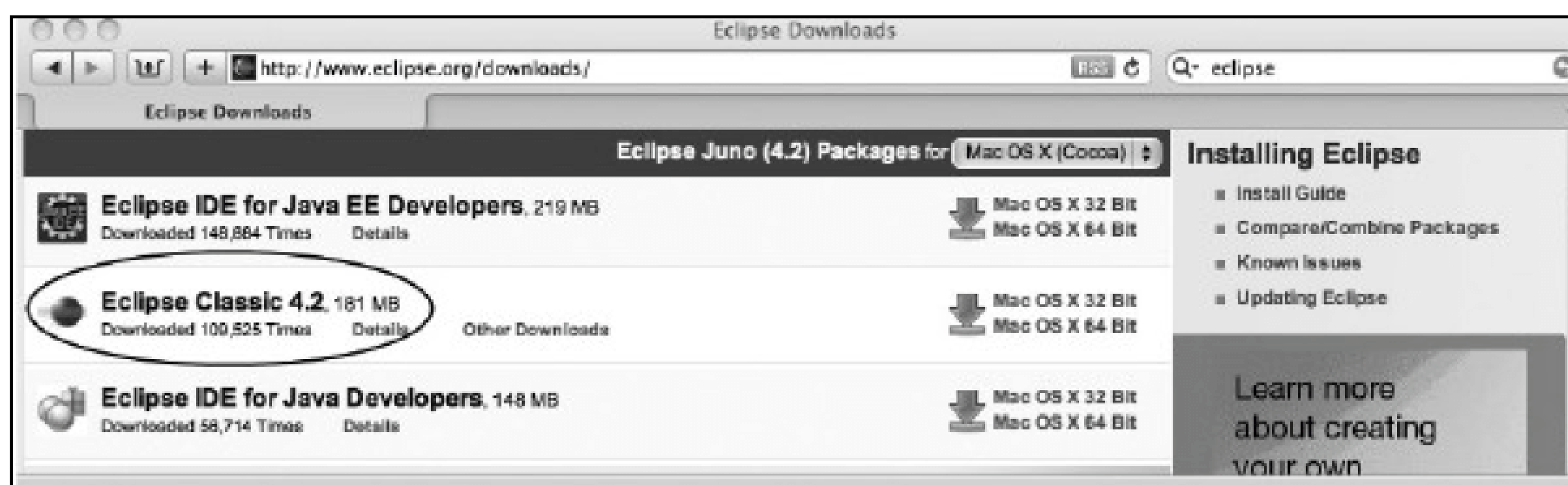


图 1-42　Eclipse 下载页面

(1) 从列表中下载 Eclipse Classic for Mac OS X 32 Bit。Eclipse 安装包以 GZIP'ed TAR 形式提供；如果使用 Safari 浏览器，文件会自动解压，但是下载后不会自动提取。

(2) 为了手动释放文件，打开一个终端窗口并进入/android 的目标目录。

(3) 执行 tar xvf ~/Downloads/eclipse-SDK-4.2-macosx-cocoa.tar，如图 1-43 所示。

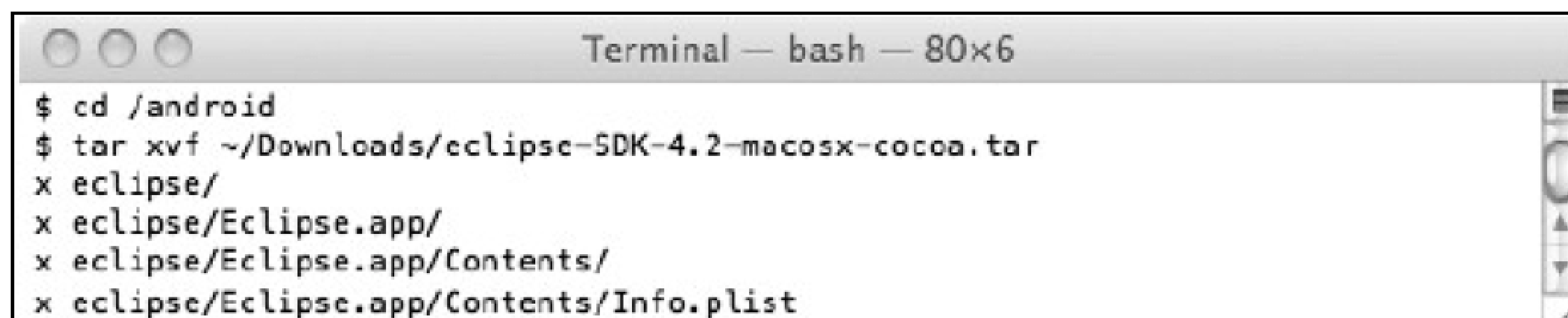


图 1-43　安装 Eclipse

为了便于访问 Eclipse，可以按照如下步骤将 Eclipse 添加到 dock 中：

(1) 进入 Eclipse 安装目录。

(2) 将 Eclipse 应用程序拖放到 Dock 中，如图 1-44 所示。

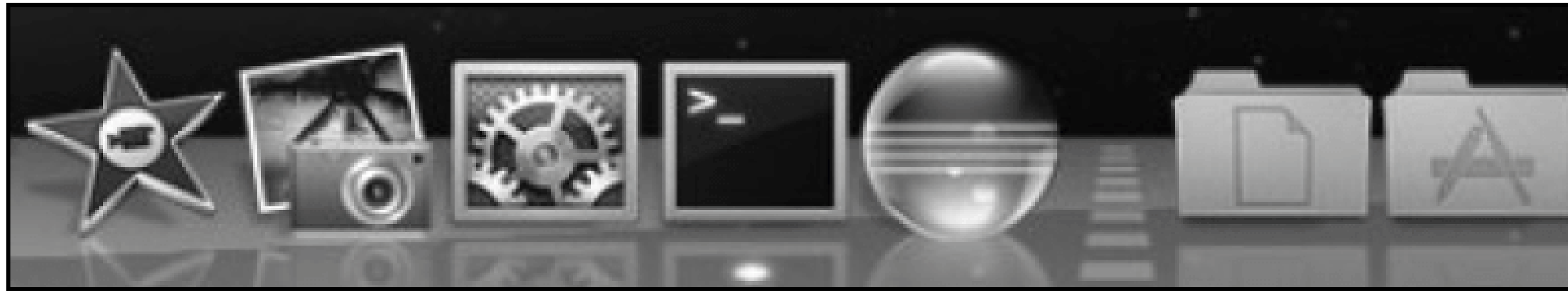


图 1-44　将 Eclipse 加入停靠栏

为了验证 Eclipse 安装结果的有效性，双击 Eclipse 图标。如果安装成功，你就会看到图 1-45 所示的 Eclipse Workspace Launcher 对话框。

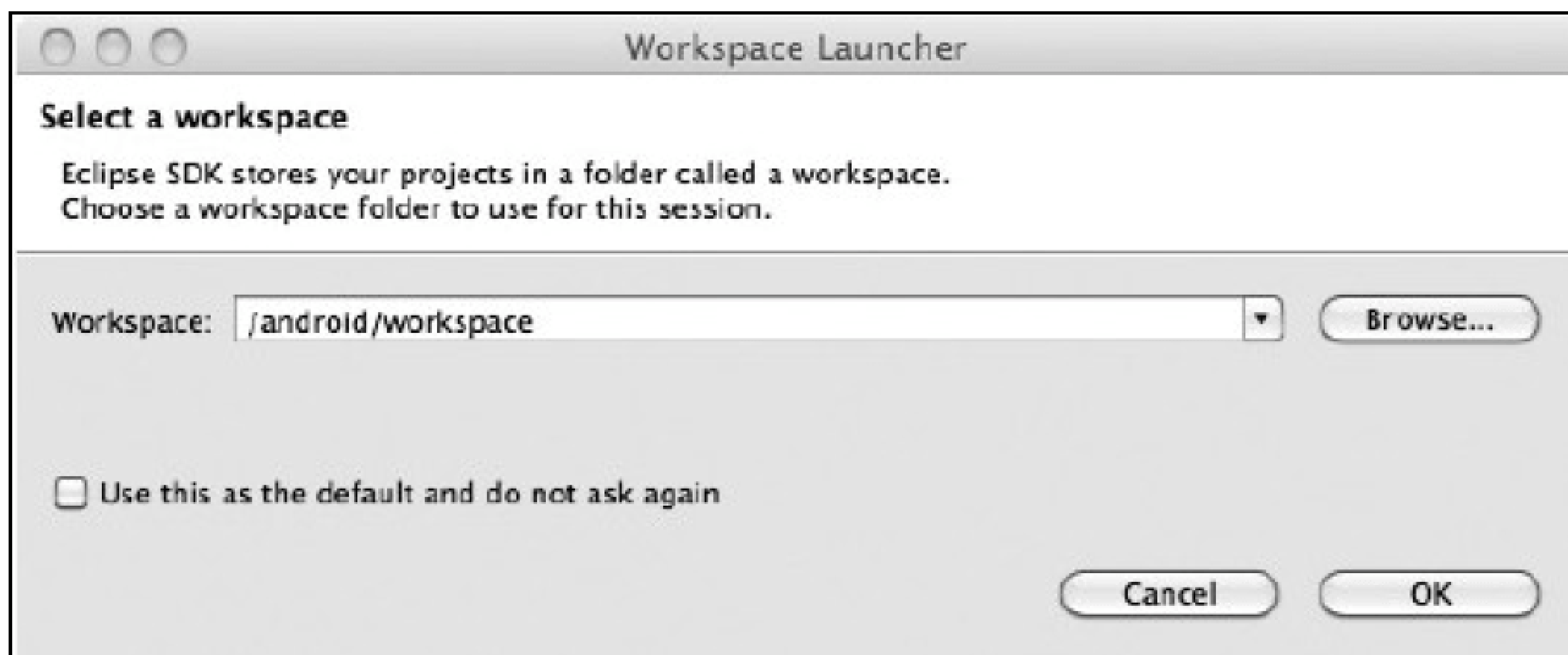


图 1-45　验证 Eclipse

1.3　Ubuntu Linux

Android 开发工具要求安装 Ubuntu Linux 8.04 32-bit 以及后续版本或者安装支持 GNU C Library(glibc)2.7 及以后版本的其他 Linux。在本节中用户需要下载并安装以下组件：

- Java JDK 6
- Apache ANT Build System
- GNU Make
- Android SDK
- Android NDK
- Eclipse IDE

1.3.1　检查 GNU C 库版本

可以通过在终端窗口中运行 ldd --version 来检查 GNU C 库版本，如图 1-46 所示。

```
cinar@onur-ubuntu: ~
$ ldd --version
ldd (Ubuntu EGLIBC 2.15-0ubuntu10) 2.15
Copyright (C) 2012 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Roland McGrath and Ulrich Drepper.
$ 
```

图 1-46　检查 GNU C 库版本

1.3.2　激活在 64 位系统上支持 32 位的功能

在 64 位 Linux 发布之后，Android 开发工具要求安装 32 位支持包。为了安装 32 位支持包，打开一个终端窗口并执行 sudo apt-get install ia32-libs-multiarch，如图 1-47 所示。

```
cinar@onur-ubuntu: ~
$ sudo apt-get install ia32-libs-multiarch
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  esound-common gtk2-engines-oxygen gtk2-engines-pixbuf libaio1 libao-common
  libao4 libaudiofile1 libcapi20-3 libesd0 libgettextpo0 libmad0 libmikmod2
  libmpg123-0 libodbc1 libopenal-data libopenal1 libqt4-designer
```

图 1-47　安装 ia32-libs-multiarch

1.3.3　在 Linux 平台上下载并安装 Java 开发工具包(JDK)

Android 开发工具要求安装 JDK 6 才能运行。不能只安装 JRE(Java Runtime Edition)，在安装 Android 开发工具之前需要先安装 Java JDK 6。除了针对 Java(gcj)GNU Compiler 以外，Android 开发工具支持多种发行版本的 JDK，例如 IBM JDK、Open JDK 以及 Oracle JDK(以前称为 Sun JDK)。由于许可问题，Oracle JDK 不能用于 Ubuntu 软件库。本书以 Open JDK 为例进行讲解。为了安装 Open JDK，打开一个终端窗口并执行 sudo apt-get install openjdk-6-jdk，如图 1-48 所示。

```
cinar@onur-ubuntu: ~
$ sudo apt-get install openjdk-6-jdk
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  icedtea-6-jre-cacao icedtea-6-jre-jamvm openjdk-6-jre openjdk-6-jre-headless
  openjdk-6-jre-lib
Suggested packages:
```

图 1-48　安装 Open JDK 6

为了验证 Open JDK 安装结果的有效性，打开一个终端窗口并在命令行方式下执行

java –version。如果安装成功，就会看到如图 1-49 所示的 Open JDK 版本号。

```
cinar@onur-ubuntu: ~
$ java -version
java version "1.6.0_24"
OpenJDK Runtime Environment (IcedTea6 1.11.1) (6b24-1.11.1-4ubuntu2)
OpenJDK Server VM (build 20.0-b12, mixed mode)
$
```

图 1-49　验证 Open JDK 的安装效果

1.3.4　在 Linux 平台上下载并安装 Apache ANT

Apache ANT 是命令行构建工具，可以驱动任何根据目标和任务描述的过程。Android 开发工具要求安装 Apache ANT 1.8 及以后版本，Apache ANT 通过 Ubuntu 软件库提供。为了安装 Apache ANT，打开一个终端窗口并执行 sudo apt-get install ant，如图 1-50 所示。

```
cinar@onur-ubuntu: ~
$ sudo apt-get install ant
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  ant-optional libxerces2-java libxml-commons-external-java
  libxml-commons-resolver1.1-java
```

图 1-50　安装 Apache ANT

为了验证 Apache ANT 的安装效果，打开一个终端窗口并在命令行方式下执行 ant –version。如果安装成功，将会显示 Apache ANT 的版本号，如图 1-51 所示。

```
cinar@onur-ubuntu: ~
$ ant -version
Apache Ant version 1.8.0 compiled on February 1 2010
$
```

图 1-51　验证 Apache ANT 安装效果

1.3.5　在 Linux 平台上下载并安装 GNU Make

GNU Make 是一种构建工具，用于控制应用程序源代码的可执行代码和其他部分代码的生成。Android NDK 要求安装 GNU Make 3.8.1 及以后版本。GNU Make 是由 Ubuntu 软件库提供的。为了安装 GNU Make，打开一个终端窗口并执行 sudo apt-get install make，如图 1-52 所示。

```
cinar@onur-ubuntu: ~
$ sudo apt-get install make
[sudo] password for cinar:
Reading package lists... Done
Building dependency tree
Reading state information... Done
make is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 225 not upgraded.
```

图 1-52　安装 GNU Make

为了验证 GNU Make 的安装效果，打开一个终端窗口并在命令行方式下执行 make –version。如果安装成功，将会显示 GNU Make 的版本号，如图 1-53 所示。

```
$ make -version
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

This program built for i686-pc-linux-gnu
$
```

图 1-53　验证 GNU Make 安装效果

1.3.6　在 Linux 平台上下载并安装 Android SDK

Android 软件开发包 (SDK) 是开发工具链的核心组件，它提供了构建、测试和调试 Android 应用程序所需要的框架 API 库和开发工具。本书编写时，Android SDK 的最新版本是 R20，请访问 http://developer.android.com/sdk/index.html 网站下载 Android SDK，如图 1-54 所示。安装步骤如下：

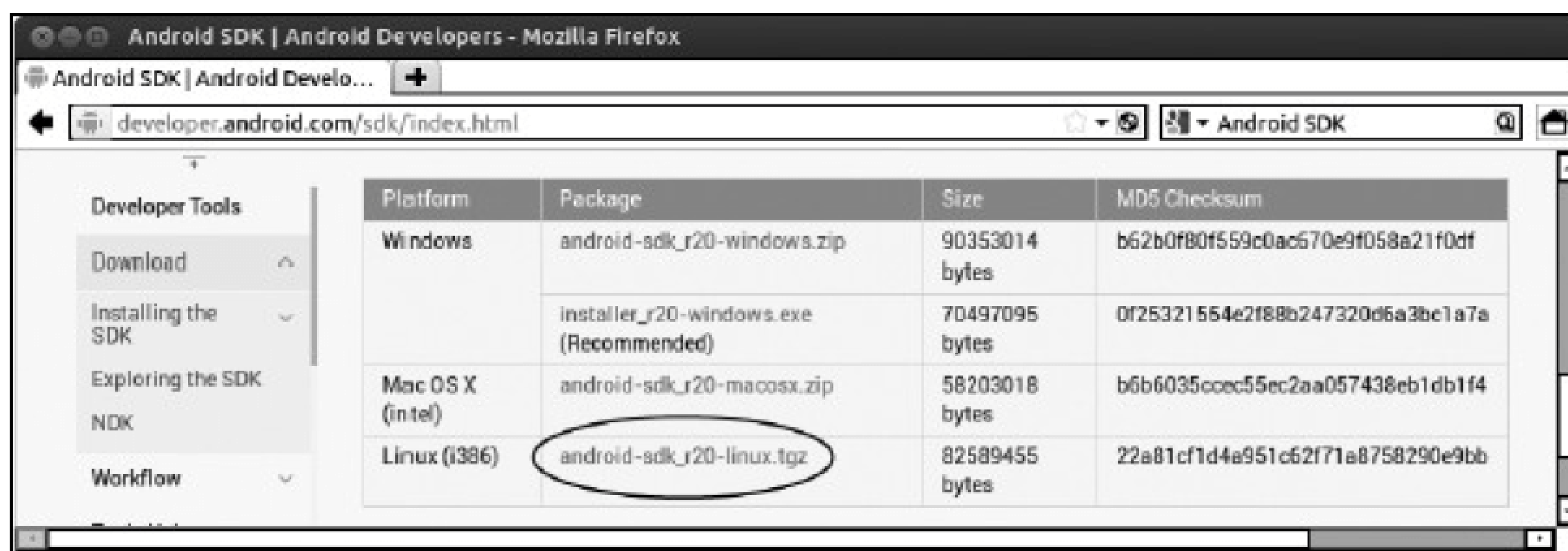


图 1-54　Android SDK 下载页面

(1) Android SDK 安装包以 GZIP'ed TAR 方式提供， 打开一个终端窗口并进入目标目录。本书中~/android 目录是保存 Android 开发工具及其相关组件的根目录。

(2) 在命令行方式下执行 tar zxvf ~/Downloads/android-sdk_r20-linux.tgz 解压 Android SDK，如图 1-55 所示。

```
$ cd ~/android
$ tar zxvf ~/Downloads/android-sdk_r20-linux.tgz
android-sdk-linux/
android-sdk-linux/platforms/
android-sdk-linux/add-ons/
android-sdk-linux/SDK Readme.txt
android-sdk-linux/tools/
```

图 1-55　安装 Android SDK

为了便于访问 Android SDK，应该把 Android SDK 二进制路径追加到系统可执行搜索路径中。假设使用 BASH shell， 打开一个终端窗口并执行下列命令(如图 1-56 所示)：

```
cinar@onur-ubuntu: ~
$ echo export ANDROID_SDK_HOME=~/android/android-sdk-linux >> ~/.bashrc
$ echo export PATH=\$ANDROID_SDK_HOME/tools:\$ANDROID_SDK_HOME/platform-tools:\$PATH
 >> ~/.bashrc
$
```

图 1-56　将 SDK 二进制路径追加到系统 PATH 变量中

- echo export ANDROID_SDK_HOME = ~/android/android-sdk-linux > > ~/.bashrc
- echo export PATH = \$ANDROID_SDK_HOME/tools:\$ANDROID_SDK_HOME/platformtools:\$PATH > > ~/.bashrc

为了验证 Android SDK 的安装效果，打开一个新的终端窗口并在命令行方式下执行 android –h。如果安装成功，将会看到图 1-57 显示的帮助信息。

```
cinar@onur-ubuntu: ~
$ android -h

       Usage:
       android [global options] action [action options]
       Global options:
  -h --help       : Help on a specific command.
  -v --verbose    : Verbose mode, shows errors, warnings and all messages.
```

图 1-57　验证 AndroidSDK 安装效果

1.3.7　在 Linux 平台上下载并安装 Android NDK

Android 原生开发工具包(NDK)是 Android SDK 的伴随工具，可以让用户用诸如 C++的原生编程语言开发 Android 应用程序。Android NDK 提供了头文件、库和交叉编译器工具链。本书编写时，Android NDK 的最新版本是 R8。请到 http://developer.android.com/tools/sdk/ndk/index.html 网站下载 Android NDK，下载部分如图 1-58 所示。下载步骤如下：



图 1-58　Android NDK 下载页面

(1) 打开终端窗口进入目标目录~/android。

(2) Android NDK 安装包以 BZIP'ed TAR 的形式提供，执行 tar jxvf ˜/Downloads/android-ndk-r8-linux-x86.tar.bz2 解压文件，如图 1-59 所示。

```
cinar@onur-ubuntu: ~
$ cd ~/android
$ tar jxvf ~/Downloads/android-ndk-r8-linux-x86.tar.bz2
android-ndk-r8/
android-ndk-r8/prebuilt/
android-ndk-r8/prebuilt/linux-x86/
android-ndk-r8/prebuilt/linux-x86/bin/
android-ndk-r8/prebuilt/linux-x86/bin/awk
```

图 1-59　安装 Android NDK

为了便于访问，要把 Android NDK 二进制路径追加到系统可执行文件搜索路径中。打开一个终端窗口并执行下列命令(如图 1-60 所示)：

```
cinar@onur-ubuntu: ~
$ echo export ANDROID_NDK_HOME=~/android/android-ndk-r8 >> ~/.bashrc
$ echo export PATH=\$ANDROID_NDK_HOME:\$PATH >> ~/.bashrc
$
```

图 1-60　将 Android NDK 二进制路径追加到系统 PATH 变量中

- `echo export ANDROID_NDK_HOME = ~/android/android-ndk-r8 > > ~/.bashrc`
- `echo export PATH = \$ANDROID_NDK_HOME:\$PATH > > ~/.bashrc`

打开一个终端窗口，在命令行方式下执行命令 ndk-build 以验证 Android NDK 的安装效果。如果安装成功，会看到 NDK 给出的关于项目目录的提示，如图 1-61 所示。

```
cinar@onur-ubuntu: ~
$ ndk-build
Android NDK: Could not find application project directory !
Android NDK: Please define the NDK_PROJECT_PATH variable to point to it.
/home/cinar/android/android-ndk-r8/build/core/build-local.mk:130: *** Android NDK: A
borting    .  Stop.
$
```

图 1-61　验证 Android NDK 安装情况

1.3.8　在 Linux 平台上下载并安装 Eclipse

Eclipse 是高度可扩展的、多语言集成开发环境。尽管原生 Android 应用程序开发不要求必须安装 Eclipse，但是 Eclipse 提供了高度集成的编码环境，它与 Android 工具结合使用从而简化应用程序的开发。本书编写时，Eclipse 的最新版本是 Juno 4.2，请访问 http://www.eclipse.org/downloads/网站下载 Eclipse，如图 1-62 所示，下载步骤如下：

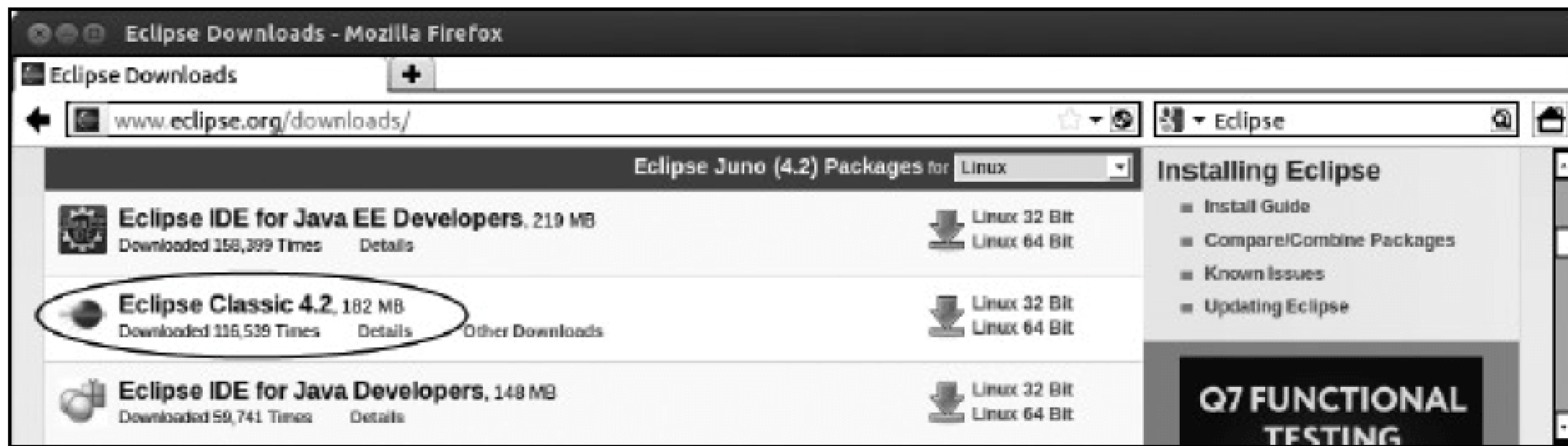


图 1-62　Eclipse 下载页面

(1) 从列表中下载 Eclipse Classic for Linux 32 Bit。

(2) 打开一个终端窗口并进入目标目录~/android。

(3) Eclipse 安装包以 GZIP'ed TAR 形式提供，通过在命令行方式下执行命令 tar xvf ~/Downloads/eclipse-SDK-4.2- linux-gtk.tar.gz 进行文件解压缩，如图 1-63 所示。



图 1-63　安装 Eclipse

为了验证 Eclipse 安装结果的有效性，进入 eclipse 目录并在命令行方式下执行命令./eclipse。如果安装成功，就会看到如图 1-64 所示的 Eclipse Workspace Launcher 对话框。

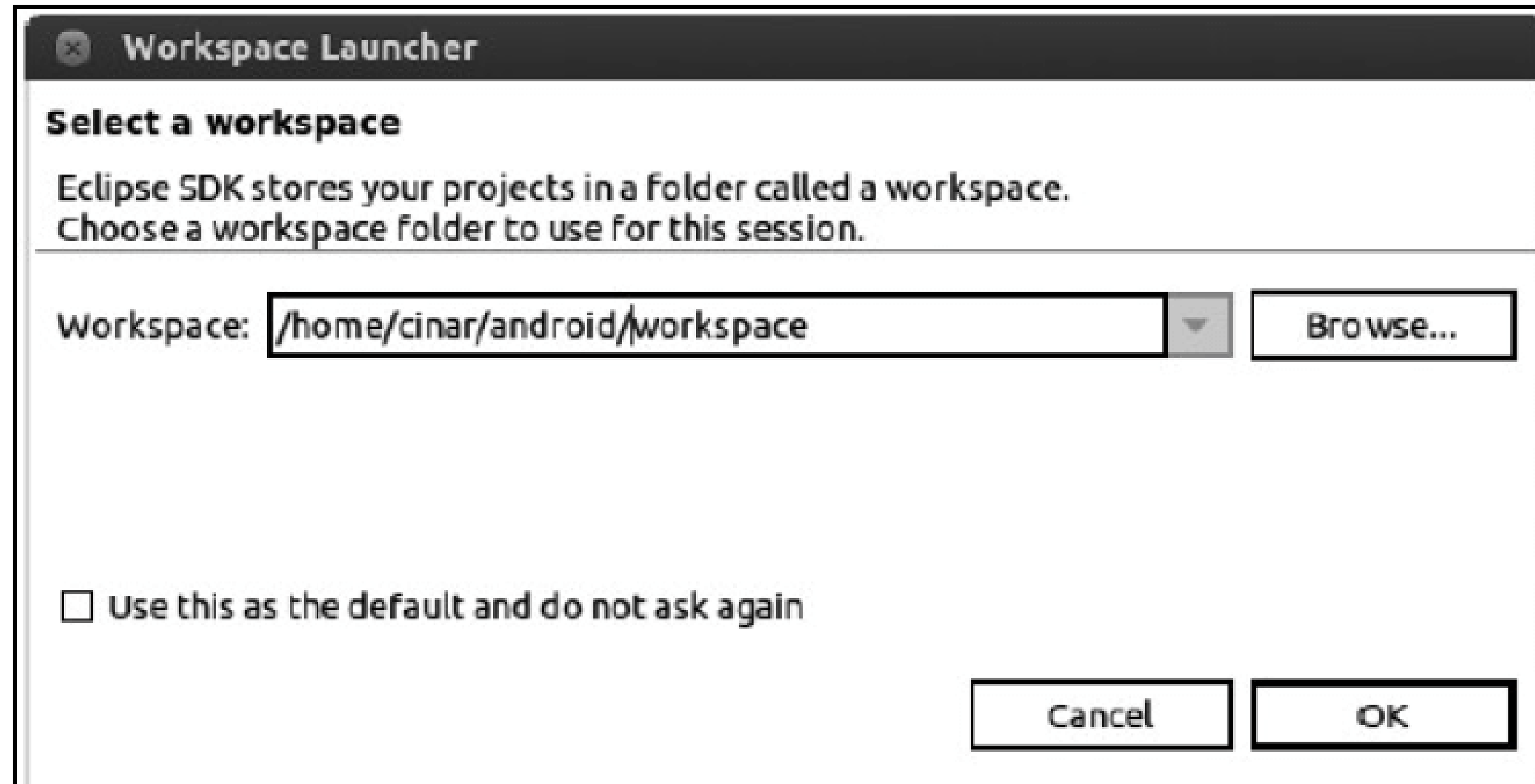


图 1-64　验证 Eclipse 安装结果

1.4　下载并安装 ADT

Android 开发工具(Android Development Tools，ADT)是 Android C++开发环境的平台独立组件，它必须安装在所有三种操作系统上。

Eclipse 平台是以插件的概念为基础构建而成，ADT 是 Eclipse 平台上用于进行 Android 应用程序开发的插件集合，ADT 是遵循开源 Apache 许可的免费软件。关于 ADT 最新版本的更多信息以及最新的安装步骤请参见 http://developer.android.com/sdk/eclipse-adt.html 网站上 Eclipse 页面的 ADT 插件部分内容。我们将使用 Eclipse 的 Install New Software 向导来安装 ADT：

(1) 在顶级菜单栏中选择 Help | Install New Software 启动安装向导，如图 1-65 所示。

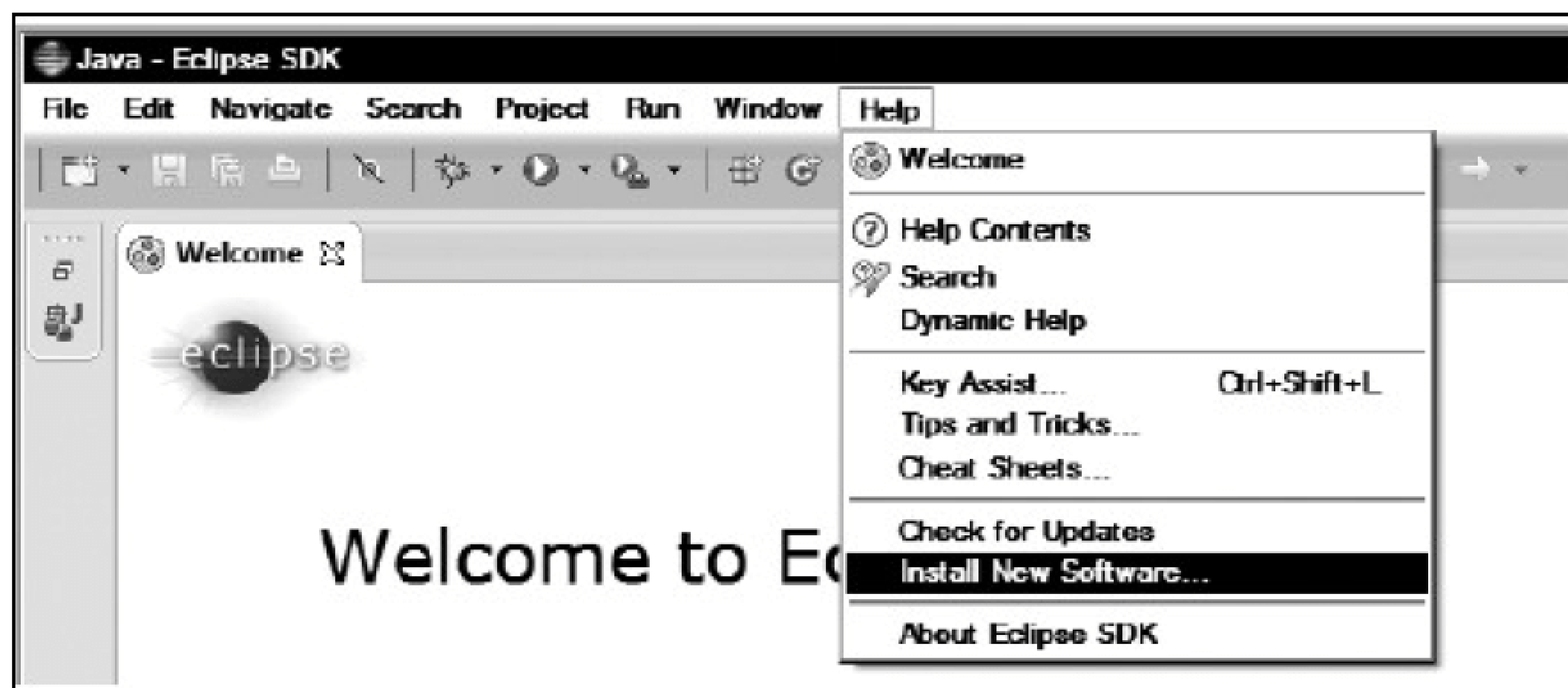


图 1-65　Eclipse 安装新软件

(2) 安装向导将启动并显示可用插件列表。因为 ADT 不是 Eclipse 官方软件库的组成部分，用户需要先添加 Android 的 Eclipse 软件库作为一个新软件站点。为完成此项任务，单击 Add 按钮，如图 1-66 所示。

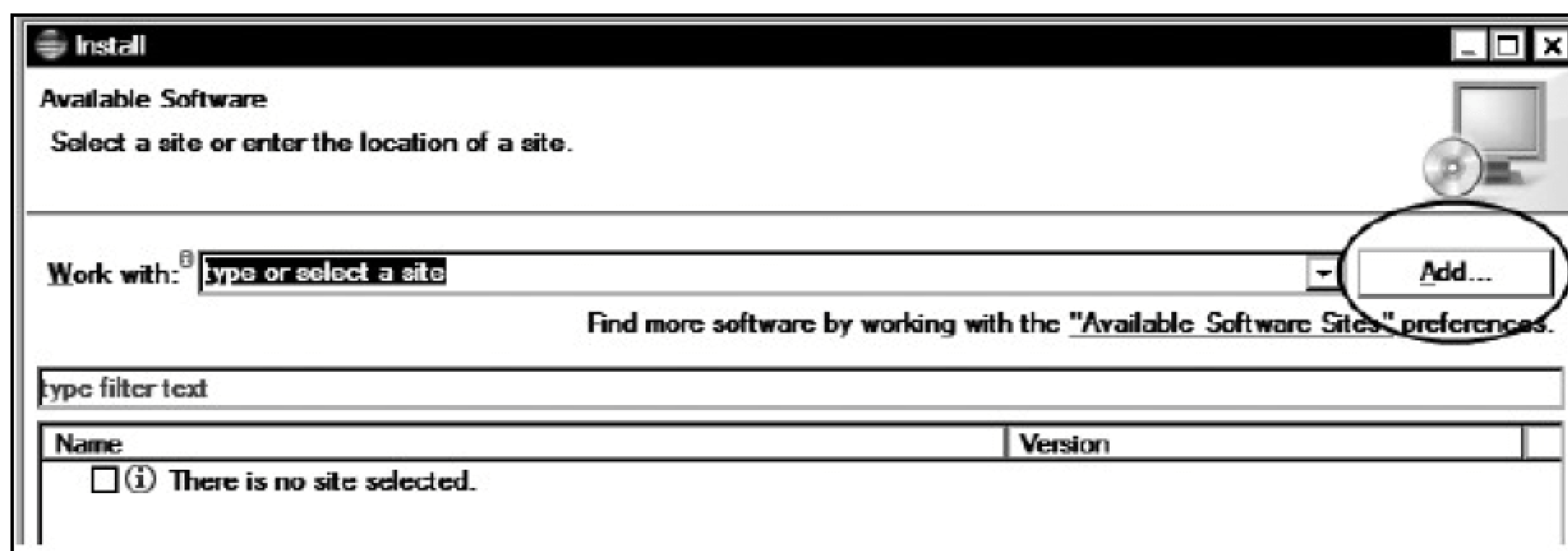


图 1-66　添加新软件库

(3) 出现 Add Repository 对话框。在 Name 字段中输入 Android ADT，并在 Location 字段中输入 Android 的 Eclipse 软件库的 URL:https://dl-ssl.google.com/android/eclipse/，如图 1-67 所示。

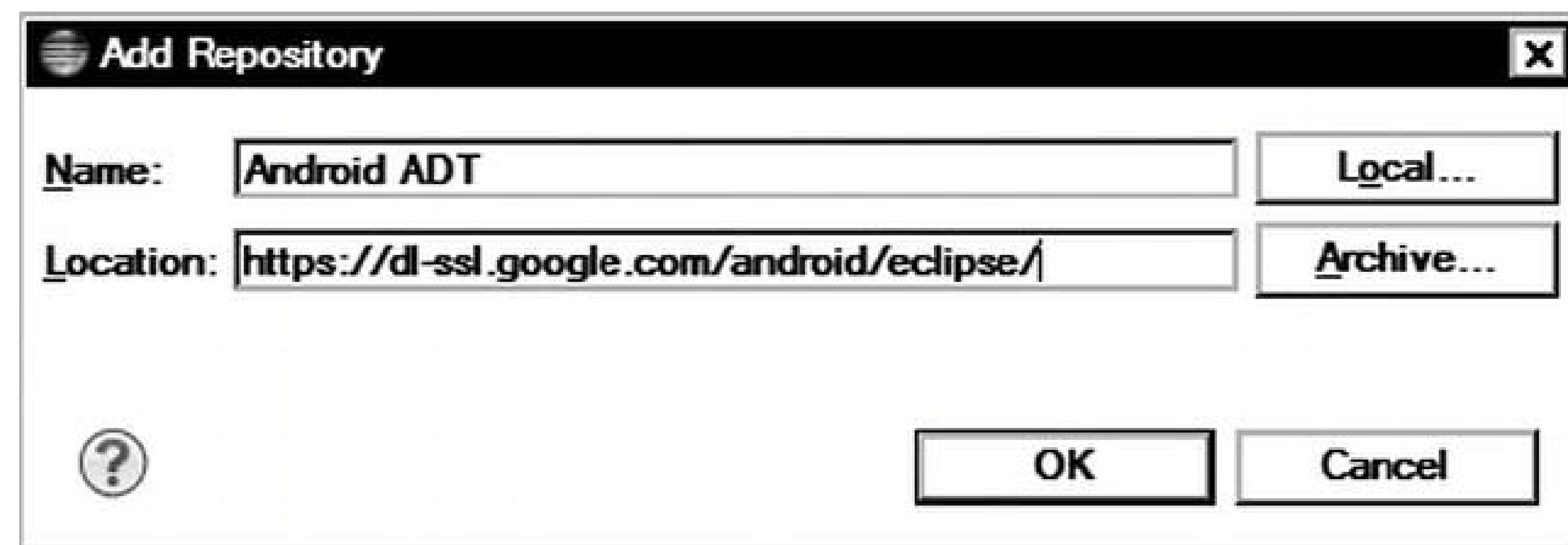


图 1-67　添加 Android ADT 软件库

(4) 单击 OK 按钮添加新软件站点。

(5) Install New Software 向导将显示可用 ADT 插件列表，如图 1-68 所示。其中的每一个插件对 Android 应用程序开发都至关重要，强烈推荐安装所有插件。

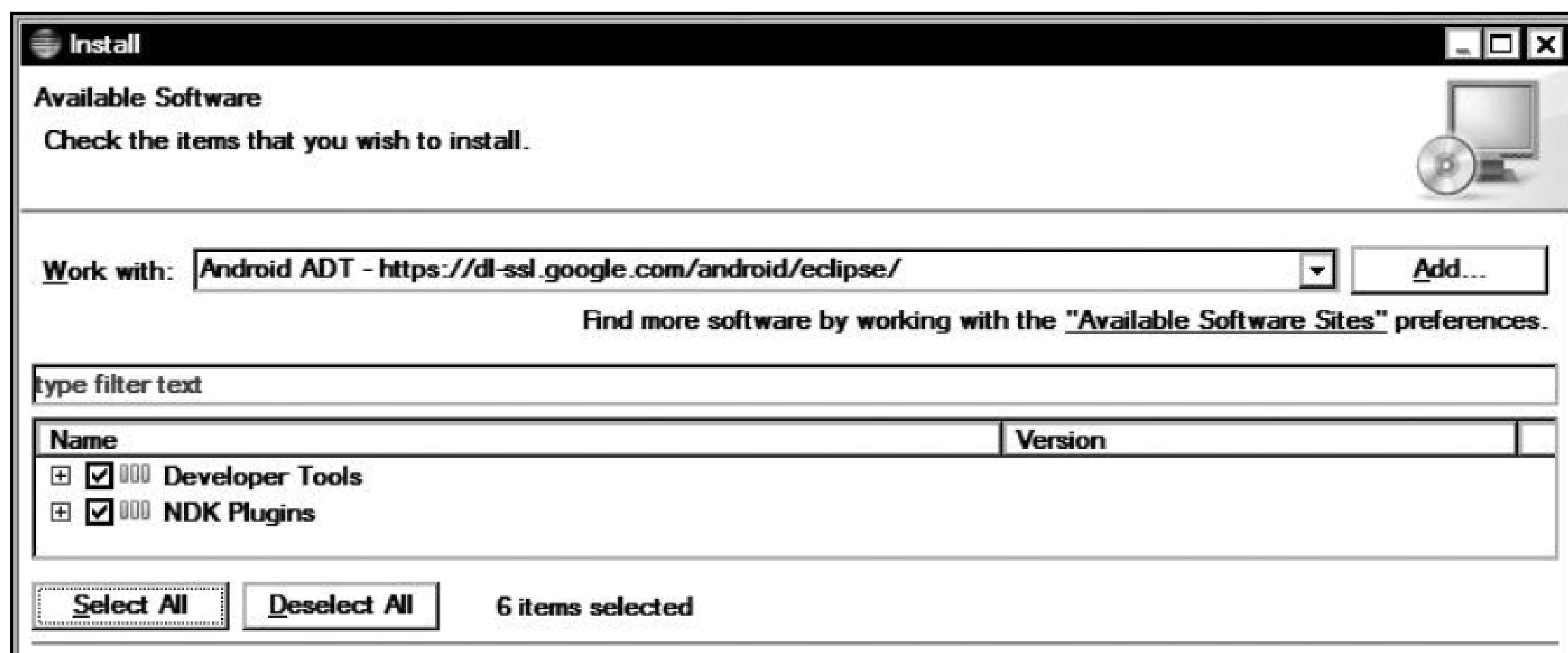


图 1-68　安装 ADT

(6) 单击 Select All 按钮选择所有的 ADT 插件。

(7) 单击 Next 按钮进入下一步操作。

(8) Eclipse 将浏览选中的插件列表以将所有的相关组件追加到列表中，然后会显示最后的下载列表让用户核查。单击 Next 按钮进入下一步操作。

(9) ADT 含有一系列遵守不同许可协议的第三方组件。在安装过程中，Eclipse 会显示每个软件的许可协议，让用户接收许可协议的内容才可以继续安装过程。选择接收许可协议，单击 Finish 按钮开始安装过程。

在未签约的 JAR 文件中的 ADT 插件会触发安全警告，如图 1-69 所示。单击 OK 按钮忽略警告并继续安装。当 ADT 插件安装完成时，Eclipse 需要重启从而使所做的修改生效。

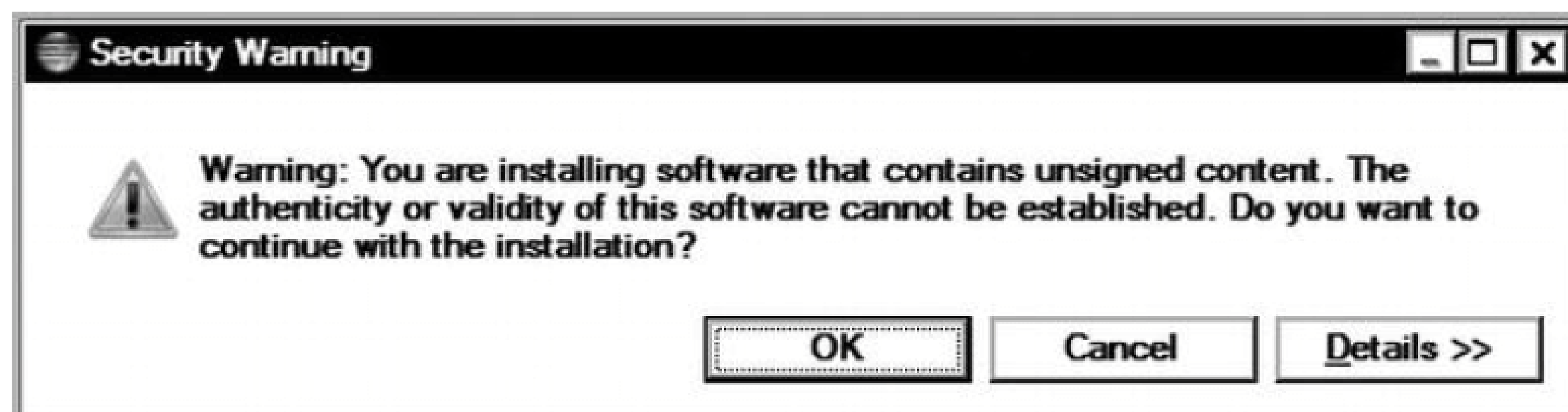


图 1-69　安全警告

重启时，ADT 将询问 Android SDK 的位置。选择 Use existing SDKs，并使用 Browse 按钮选择 Android SDK 的安装目录，如图 1-70 所示。

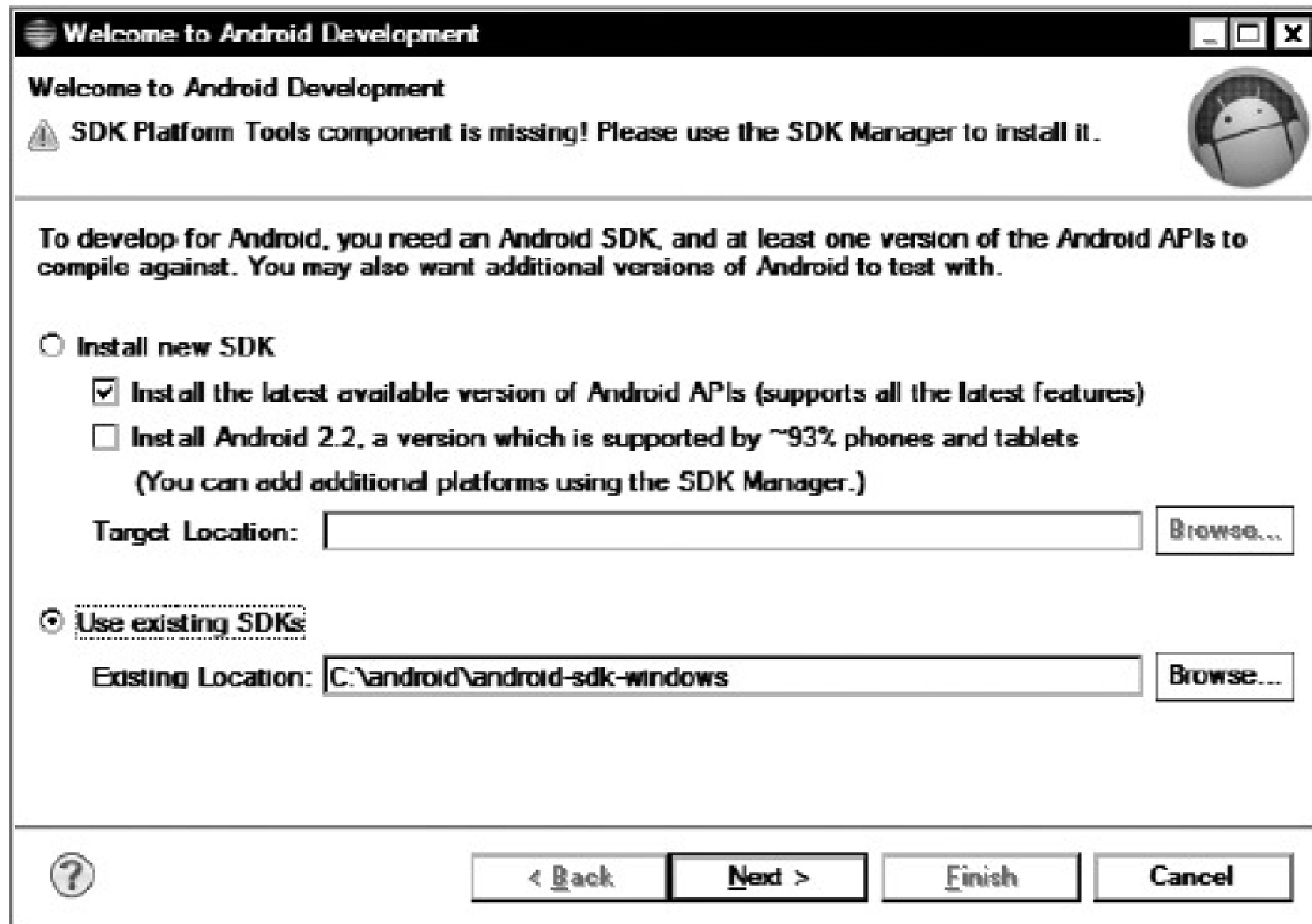


图 1-70　选择 Android SDK 位置

单击 Next 按钮进入下一步操作。

1.4.1　安装 Android 平台包

选择了 Android SDK 位置之后，ADT 验证 Android SDK 和 Android Platform 包。Android SDK 安装程序只包含 Android 开发工具，Android Platform 包需要单独安装，这样才能构建 Android 应用程序。验证完成后，显示 SDK 验证警告对话框，如图 1-71 所示。

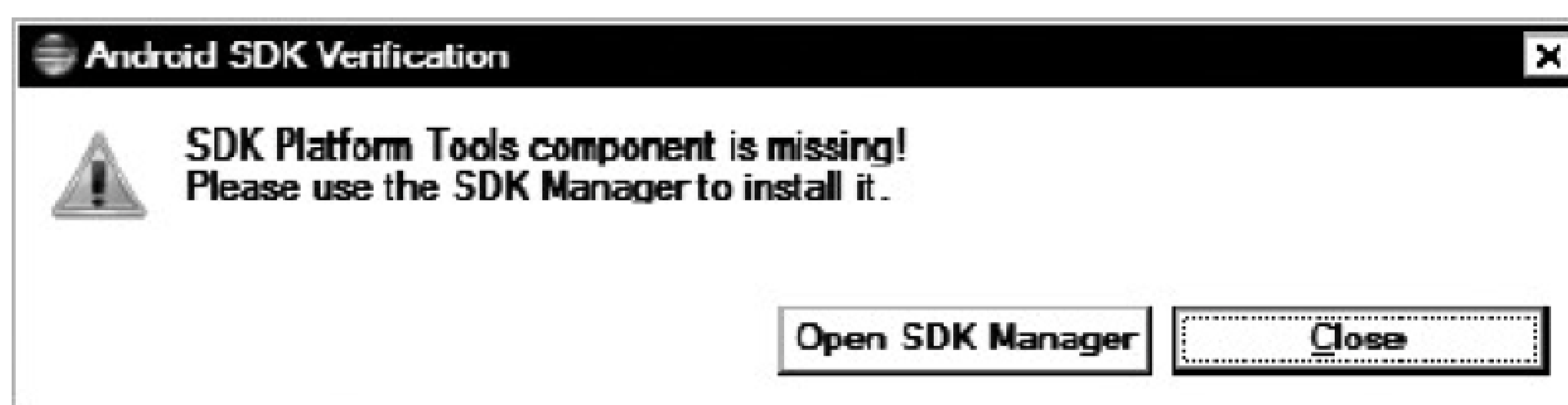


图 1-71　ADT Android SDK 验证

单击 Open SDK Manager 按钮启动 Android SDK Manager，按照图 1-72 所示的步骤进行操作。

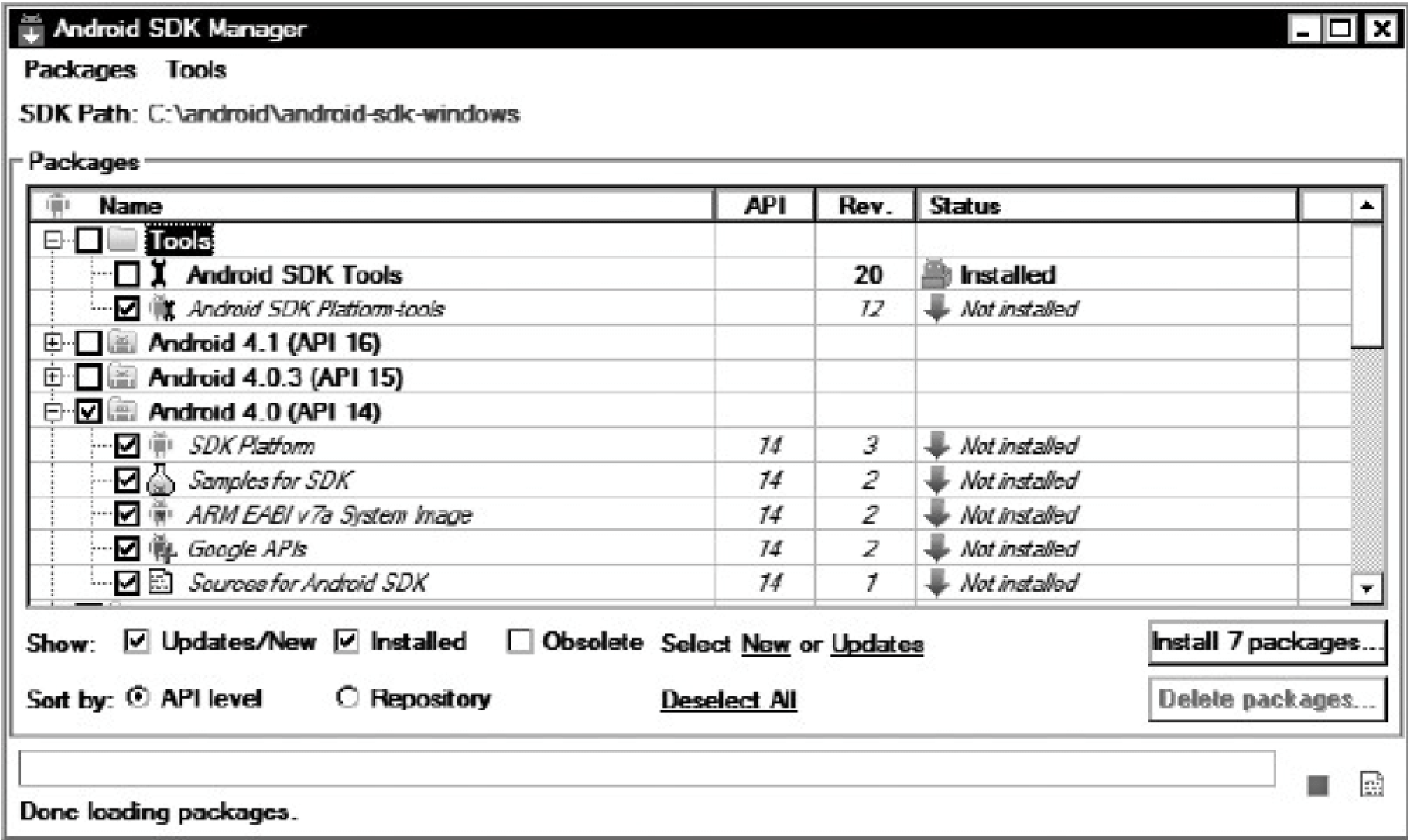


图 1-72　Android SDK 管理器

(1) 从可用包列表中展开 Tools 目录，并选择 Android SDK Platform-Tools。

(2) 选择 Android 4.0 (API 14)类别。

(3) 单击 Install *N* Packages 按钮开始安装。

Android SDK 管理器将显示选中包的许可协议，接受所有的许可协议继续安装。

1.4.2 配置模拟器

Android SDK 带有一个功能齐全的模拟器，该模拟器是一个在用户的机器上运行的虚拟设备。Android 模拟器可以让你在机器上本地开发和测试 Android 应用程序，而不需要使用物理设备。

Android 模拟器运行一个包括 Linux 内核的完整 Android 系统栈。它是可以模仿真实设备的所有硬件和软件特性的完全虚拟的设备，每一个特征都可以用 Android 虚拟设备管理器(Android Virtual Device，AVD)来定制。如图 1-73 所示，启动 AVD Manager 主菜单中选择 Window | AVD Manager。

单击 AVD Manager 对话框右侧的 New 按钮定义一个新的模拟器配置，如图 1-74 所示。

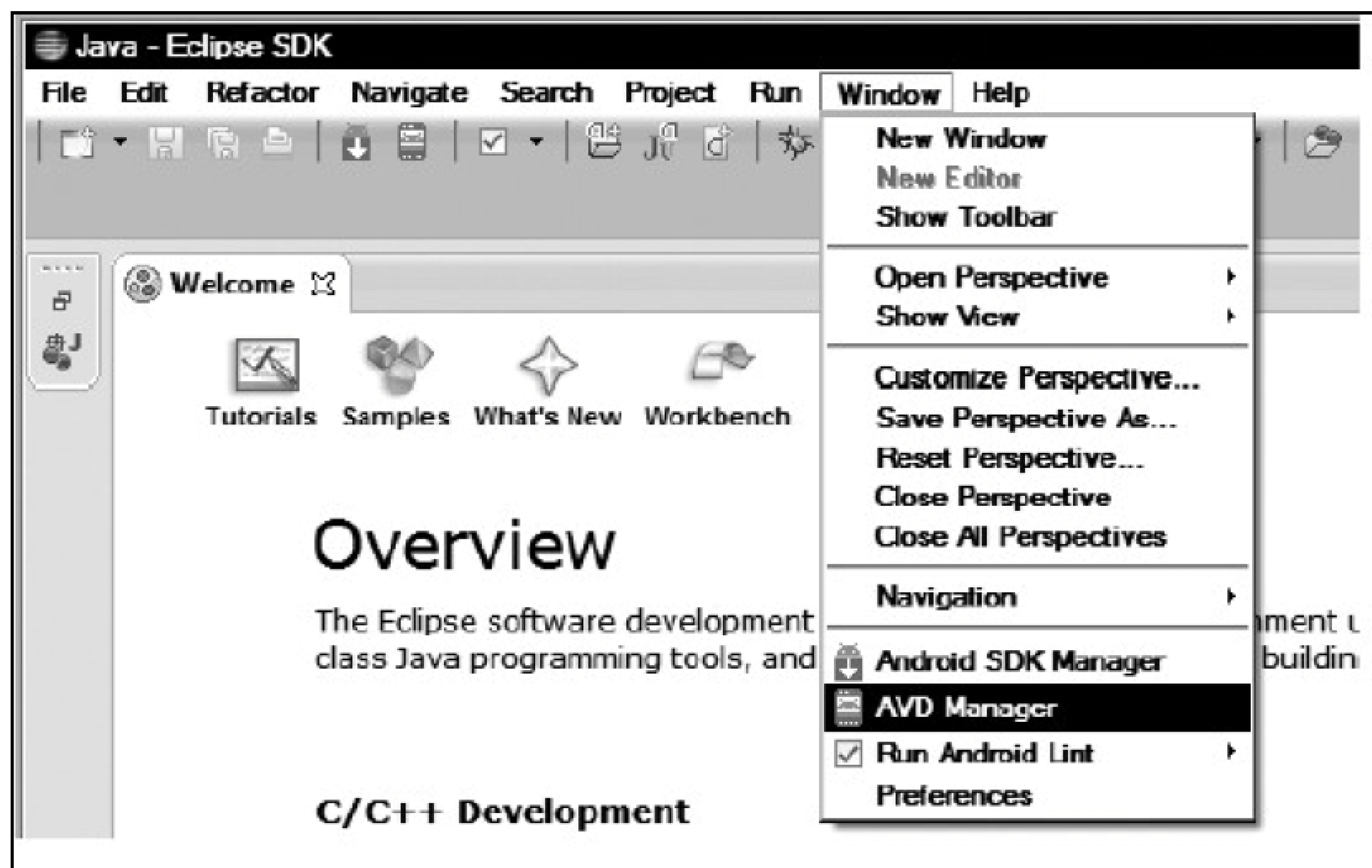


图 1-73　AVD Manager 菜单

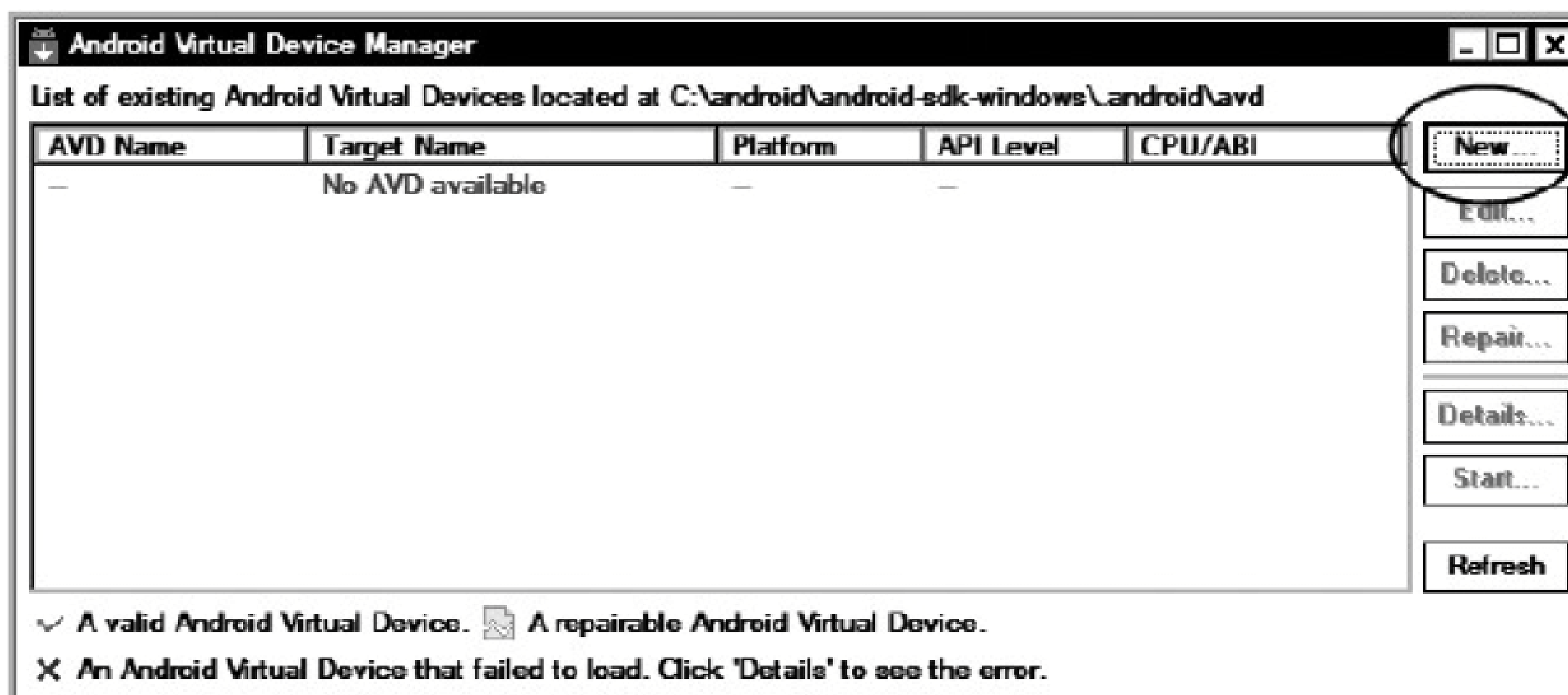


图 1-74　AVD Manager

在本书中，通常在完成了资源配置之后使用 Android 模拟器。推荐使用下面的虚拟机配置执行本书中的示例代码，按照下面的说明设置各字段值，如图 1-75 所示。

- Name 参数应设置为 Android_14。
- Target 参数应设置为 Android 4.0 – API Level 14。
- SD Card 的大小应设置为至少 128 MB。

其他参数可以使用默认值。

为了验证新定义的模拟器配置，打开 AVD Manager，从列表中选择所配置的模拟器名称。单击 Start 按钮启动模拟器实例。如果配置成功，将会出现模拟器(如图 1-76)所示。

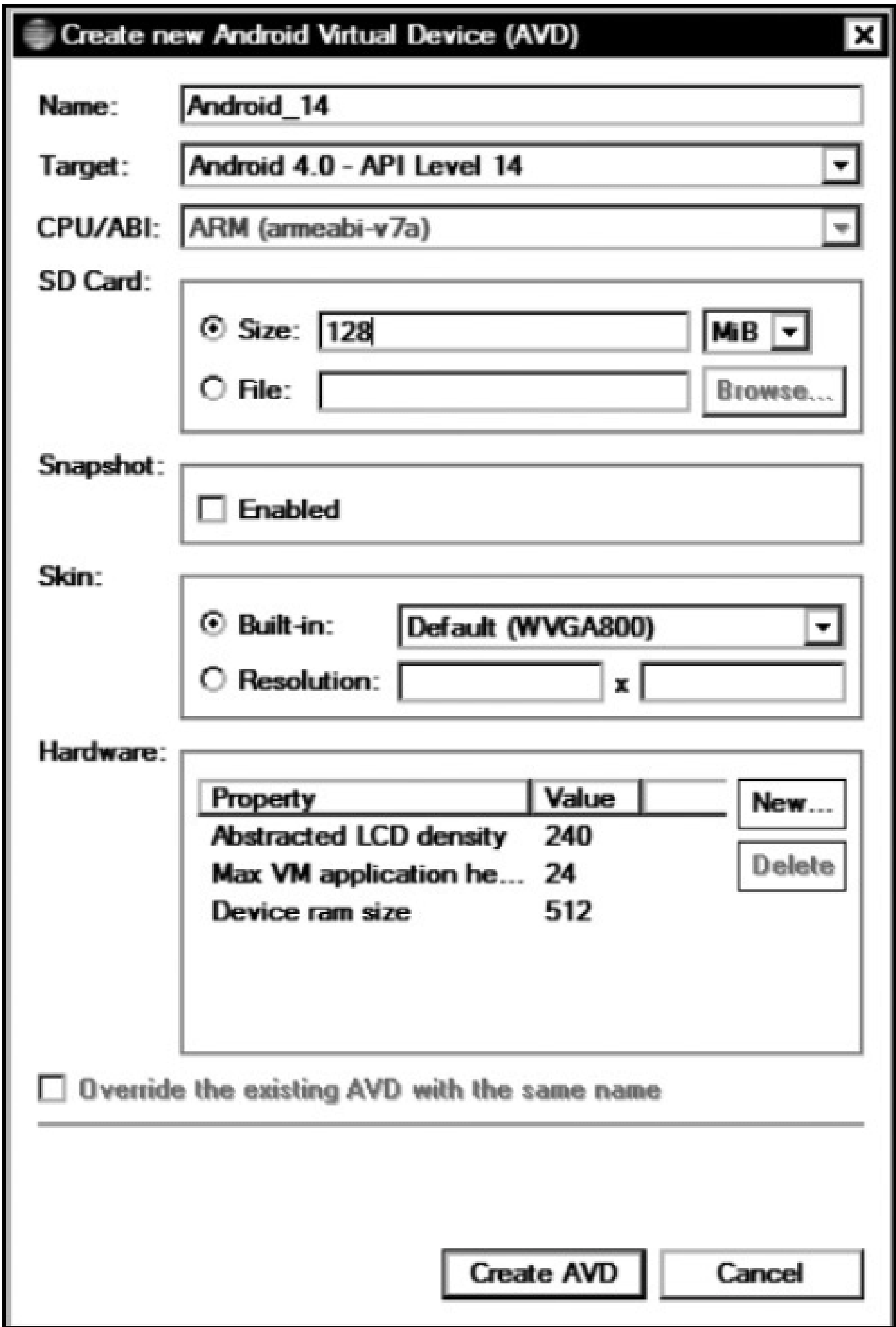


图 1-75　新建模拟器配置



图 1-76　新定义的模拟器配置运行

1.5　小结

本章通过在目标操作系统上安装 Android 开发工具及其相关组件配置你的 Android C++开发环境，定义了 Android 模拟器配置以执行以后章节中将出现的示例代码，第 2 章将详细介绍 Android NDK。

第 2 章

深入了解 Android NDK

在第 1 章中我们通过安装 Android 开发工具及其相关工具配置了开发环境。在这些工具中，Android 原生开发包(NDK)将用于 Android 平台上的 C++开发。Android NDK 是 Android 软件开发包(SDK)的相关工具集，用来扩展 Android SDK 的功能，从而使开发人员能够使用机器代码生成的编程语言(如 C、C++和汇编语言)实现一些对代码性能要求较高的模块并将这些模块嵌入到 Android 应用程序中。

本章开始深入探讨 Android NDK。我们将使用 Android NDK 自带的 hello-jni 示例程序来学习 Android NDK 的构建系统。

2.1 Android NDK 提供的组件

Android NDK 不是一个单独的工具；它是一个包含 API、交叉编译器、链接程序、调试器、构建工具、文档和示例应用程序的综合工具集。以下是 Android NDK 的一些主要组件：

- ARM、x86 和 MIPS 交叉编译器
- 构建系统
- Java 原生接口头文件
- C 库
- Math 库
- POSIX 线程
- 最小的 C++库
- ZLib 压缩库
- 动态链接库
- Android 日志库
- Android 像素缓冲区库

- Android 原生应用 APIs
- OpenGL ES 3D 图形库
- OpenSL ES 原生音频库
- OpenMAX AL 最小支持

2.2 Android NDK 的结构

在安装过程中，所有的 Android NDK 组件都被安装在目标目录下。下面介绍一些重要文件和子目录。

- **ndk-build**：该 shell 脚本是 Android NDK 构建系统的起始点。本章将在深入学习 Android NDK 构建系统的同时详细阐述 ndk-build。
- **ndk-gdb**：该 shell 脚本允许用 GUN 调试器调试原生组件。第 5 章讨论原生组件调试时将详细阐述 ndk-gdb。
- **ndk-stack**：该 shell 脚本可以帮助分析原生组件崩溃时的堆栈追踪。第 5 章讨论原生组件的故障排除和故障分析时将详细阐述 ndk-stack。
- **build**：该目录包含了 Android NDK 构建系统的所有模块。本章将详细介绍 Android NDK 构建系统。
- **platforms**：该目录包含了支持不同 Android 目标版本的头文件和库文件。Android NDK 构建系统会根据具体的 Android 版本自动引用这些文档。
- **samples**：该目录包含了一些示例应用程序，这些程序可以体现 Android NDK 的性能。示例项目对于学习如何使用 Android NDK 的特性很有帮助。
- **sources**：该目录包含了可供开发人员导入到现有的 Android NDK 项目的一些共享模块。
- **toolchains**：该目录包含目前 Android NDK 支持的不同目标机体系结构的交叉编译器。Android NDK 目前支持 ARM、X86 和 MIPS 机体系结构。Android NDK 构建系统根据选定的体系结构使用不同的交叉编译器。

Android NDK 最重要的组件是它的构建系统，它包含了所有的其他组件。想要更好地了解构建系统的工作原理，先看一个示例。

2.3 以一个示例开始

下面以 Android NDK 自带的 hello-jni 示例应用程序开始讲解。之后可以通过修改它来展示 Android NDK 构建系统所提供的不同功能，例如：

- 建立一个共享库
- 建立多种共享库
- 建立静态库
- 利用共享库共享通用模块
- 在多种 NDK 项目间共享模块

- 使用预建库
- 建立独立的可执行文件
- 其他构建系统变量和宏
- 定义新变量和条件操作

打开第 1 章安装好的 Eclipse IDE。虽然 Android NDK 不要求必须使用 IDE，但使用 IDE 可以帮助用户直观地检查项目结构和构建流。在启动阶段，Eclipse 会让你选择工作区；可以选择默认值并继续。

2.3.1　指定 Android NDK 的位置

因为是首次用工作区进行 Android NDK 开发，所以需要指定 Android NDK 的位置。

(1) 在 Windows 和 Linux 平台上，在主菜单栏中选择 Preference 菜单项。在 Mac OS X 平台上，使用 Eclipse 中的应用程序菜单并且选择 Preferences 菜单项。

(2) 如图 2-1 所示，Preferences 对话框左边的窗格包含了一个树状的 preference 分类列表。展开 Android 然后在树状列表中选择 NDK。

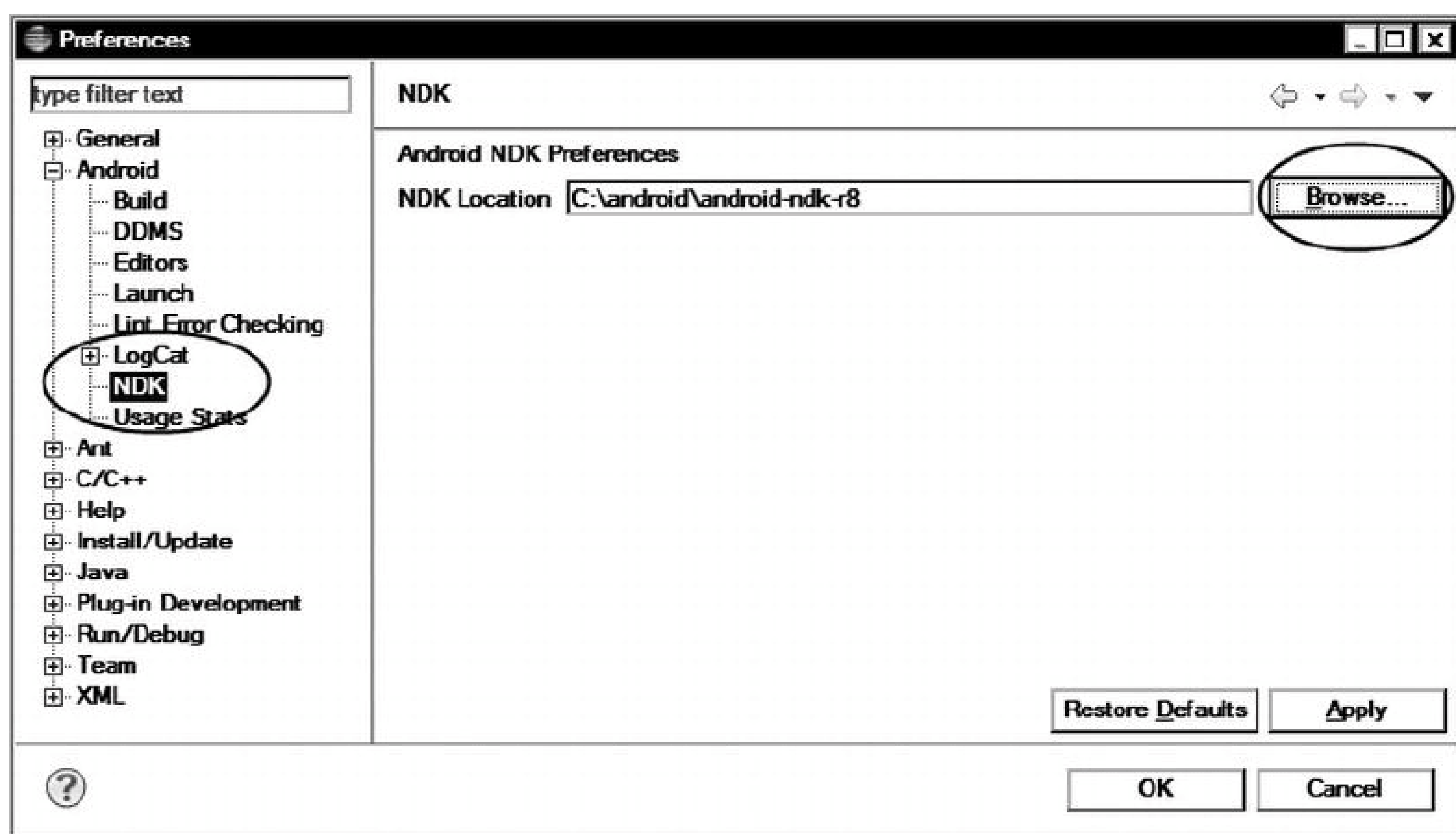


图 2-1　选择的 Android NDK 位置

(3) 在右边的窗格中，单击 Browse 按钮，并利用文件浏览器选择 Android NDK 安装位置。

选择的 NDK 位置仅对当前 Eclipse 工作区有效。如果以后要用其他工作区，需要重复以上过程。

2.3.2　导入示例项目

上节已经讲过，在 samples 目录下包含了 Android NDK 安装程序自带的示例应用程序。现在可以使用其中的示例应用程序。

在主菜单栏中选择 File，然后选择 Import 菜单项打开 Import 向导。在导入资源列表中，展开 Android 并选择 Existing Android Code into Workspace，如图 2-2 所示。单击 Next 按钮

进行下一步。

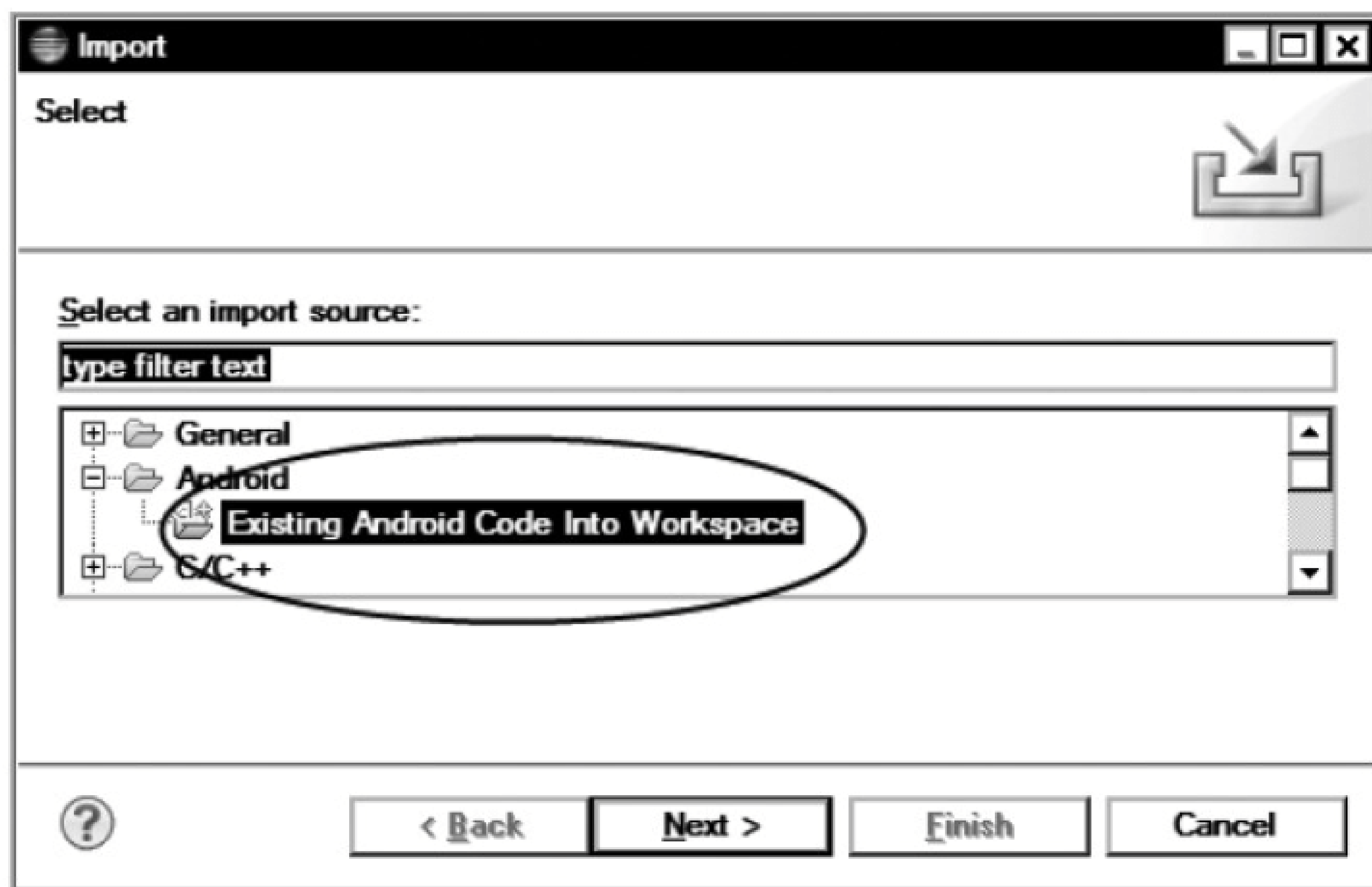


图 2-2　将现有的 Android 代码导入到工作区中

如图 2-3 所示，使用 Browse 按钮打开文件资源管理器并进入<Android NDK>/samples/hello-jni 目录。hello-jni 项目是一个简单的“Hello World”的 Android NDK 项目。项目目录包含实际项目和测试项目。为了简化问题，先不选择测试项目，只选主项目。不对 Android NDK 安装目录做任何修改是保证安全的正确做法。选中 Copy projects into workspace 选项让 Eclipse 将项目代码复制到工作区，这样就可以对副本而不是原始项目进行操作。单击 Next 按钮开始将项目导入工作区。

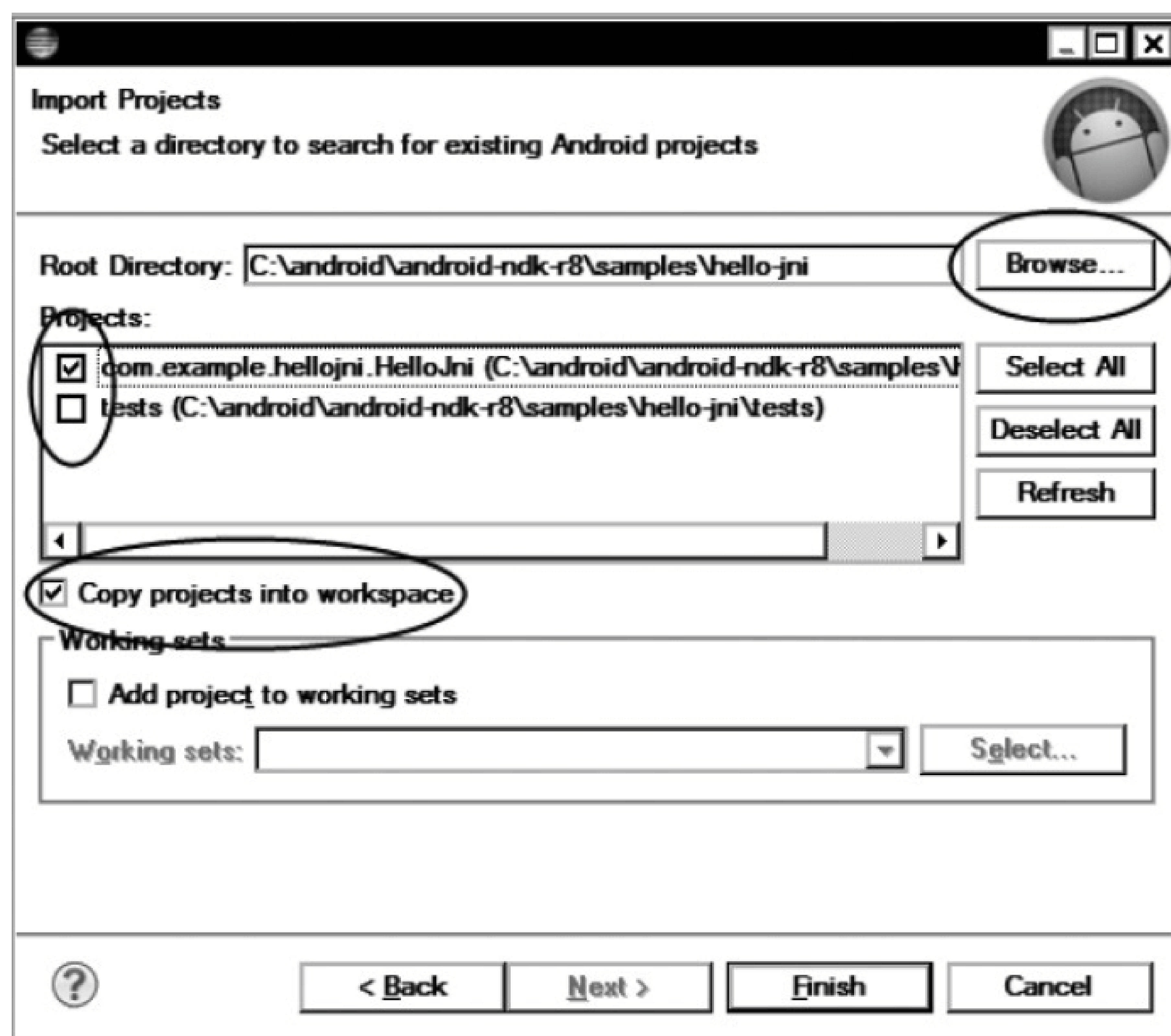


图 2-3　导入 hello-jni Android NDK 项目

如图 2-4 所示，在导入过程的最后一步，控制台会出现一个错误信息。回顾第 1 章的内容，那里只用 SDK Manager 下载了适用于 Android 4.0(API Level 14)平台的 APIs。而 hello-jni 项目是基于 Android 1.5(API Level 3)开发的。

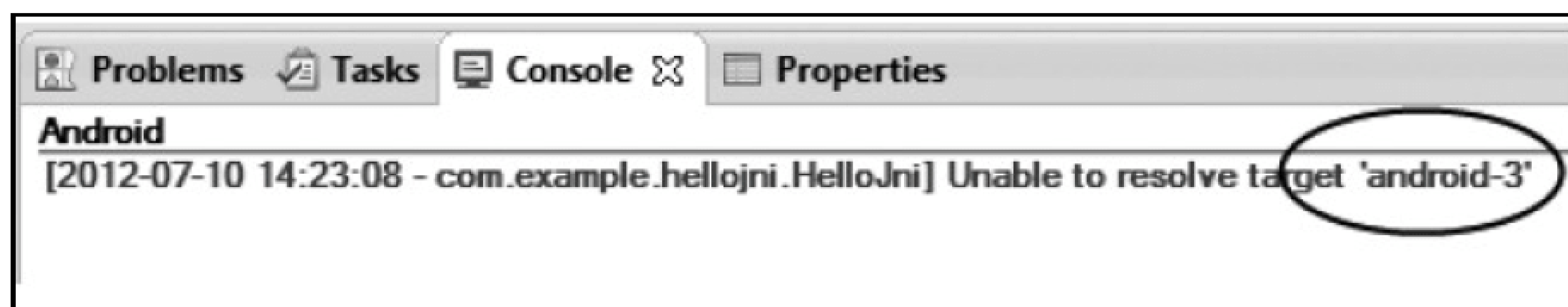


图 2-4 不能解决目标 API Level 3

API Levels 是向后兼容的，所以不用下载 API Level 3，而是在 Eclipse 的 Project Explorer 视图中右击 com.example. hellojni. HelloJni 项目，在上下文菜单中选择 Properties 打开项目属性对话框。项目属性对话框左边的窗格包含了一个树状的项目属性分类列表。在树状列表中选择 Android，并在右边窗格中选择 Android 4.0 作为项目构建目标(如图 2-5 所示)。

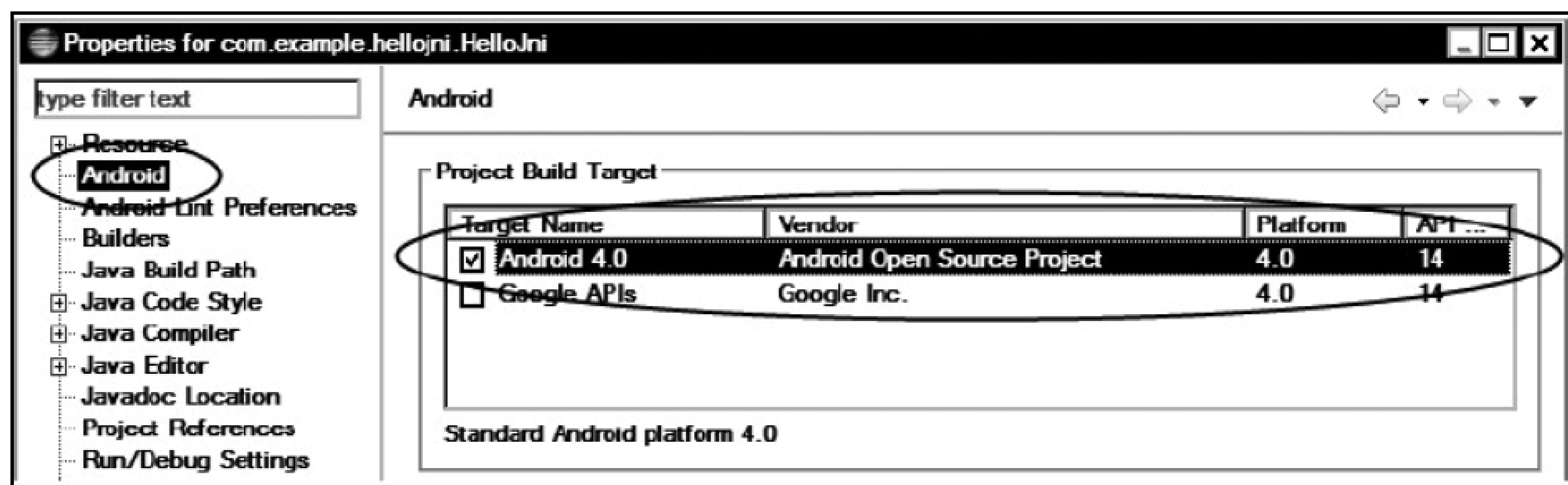


图 2-5 将 Android 4.0 选为项目构建目标

单击 OK 按钮确认上述更改。Eclipse 会用已选的项目构建目标来重建项目。

2.3.3 向项目中添加原生支持

Import Android Project 向导只将项目作为 Android Java 项目导入。为了让构建流包含原生组件，需要手动添加原生支持。在 Eclipse 的 Project Explorer 视图中右击 com.example. hellojni.HelloJni 项目，鼠标停在 Android Tools 菜单项上并在上下文菜单中选择 Add Native Support。打开 Add Android Native Support 对话框，如图 2-6 所示。由于该项目已经包含了一个原生项目，所以库名可以保持不变，单击 Finish 按钮继续。

如果是第一次向 Java-only 项目中添加原生支持，可以在该对话框中指定首选的共享库名，在将构建文件自动生成为进程的一部分时会使用该首选共享库名称。

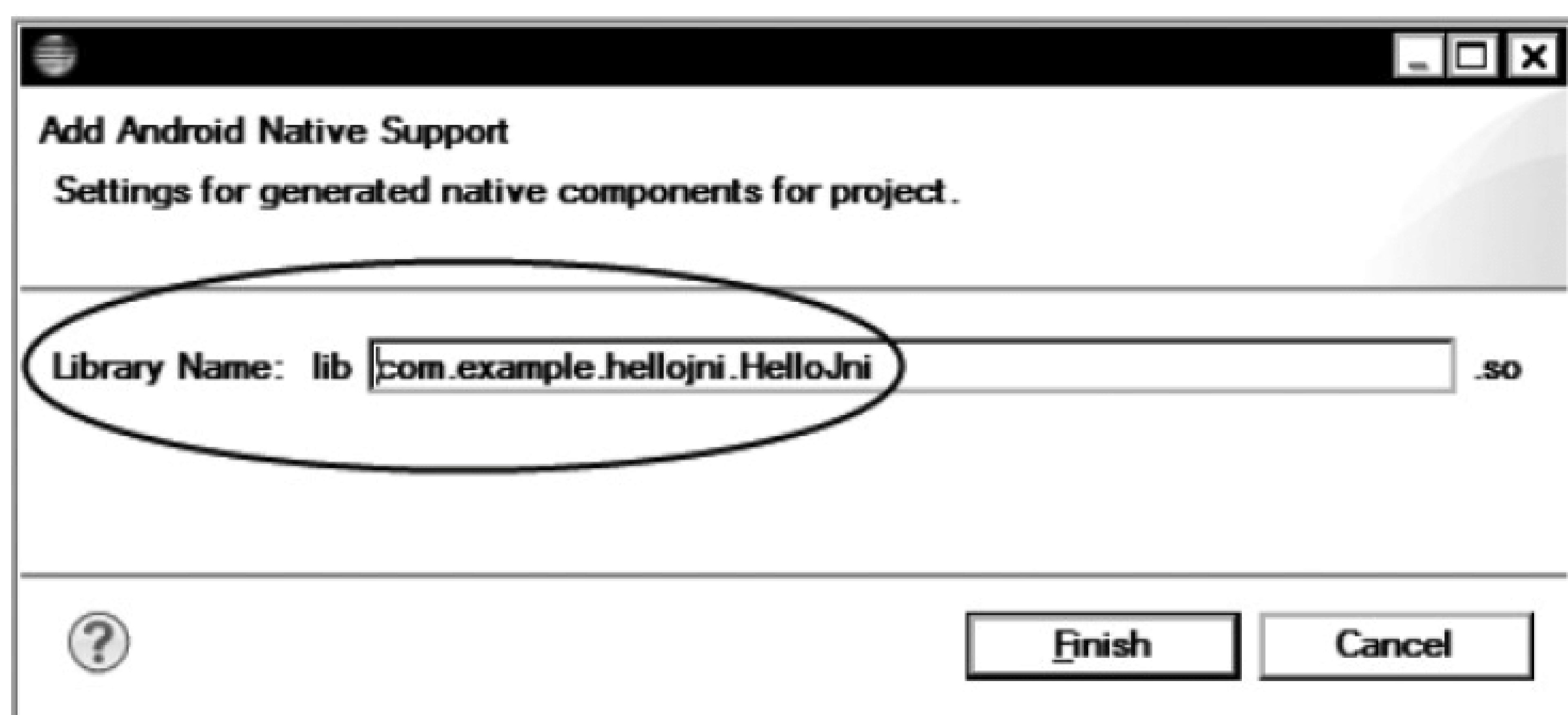


图 2-6 添加 Android 原生支持

2.3.4 运行项目

现在已经完成了项目的编写，可以在 Android 模拟器上运行该项目。在主菜单上选择 Run，然后在子菜单中选择 Run。因为是第一次运行该项目，Eclipse 会通过 Run As 对话框让用户选择该项目的运行方式。在列表中选择 Android Application 并单击 OK 按钮继续。将会启动 Android 模拟器；Eclipse 会自动部署并执行该项目，如图 2-7 所示。Android 模拟器是一个虚拟机，它完全启动 Android 操作系统可能需要花几分钟的时间。



图 2-7 运行原生项目的 Android 模拟器

或许你已经注意到：运行该项目的过程和运行 Java-only 项目的过程一模一样。因为向项目自动添加原生支持的同时已经在用户不知道的情况下将必要步骤包含在构建过程中了。仍然可以在 Console 视图中查看 Android NDK 构建系统发来的消息，如图 2-8 所示。

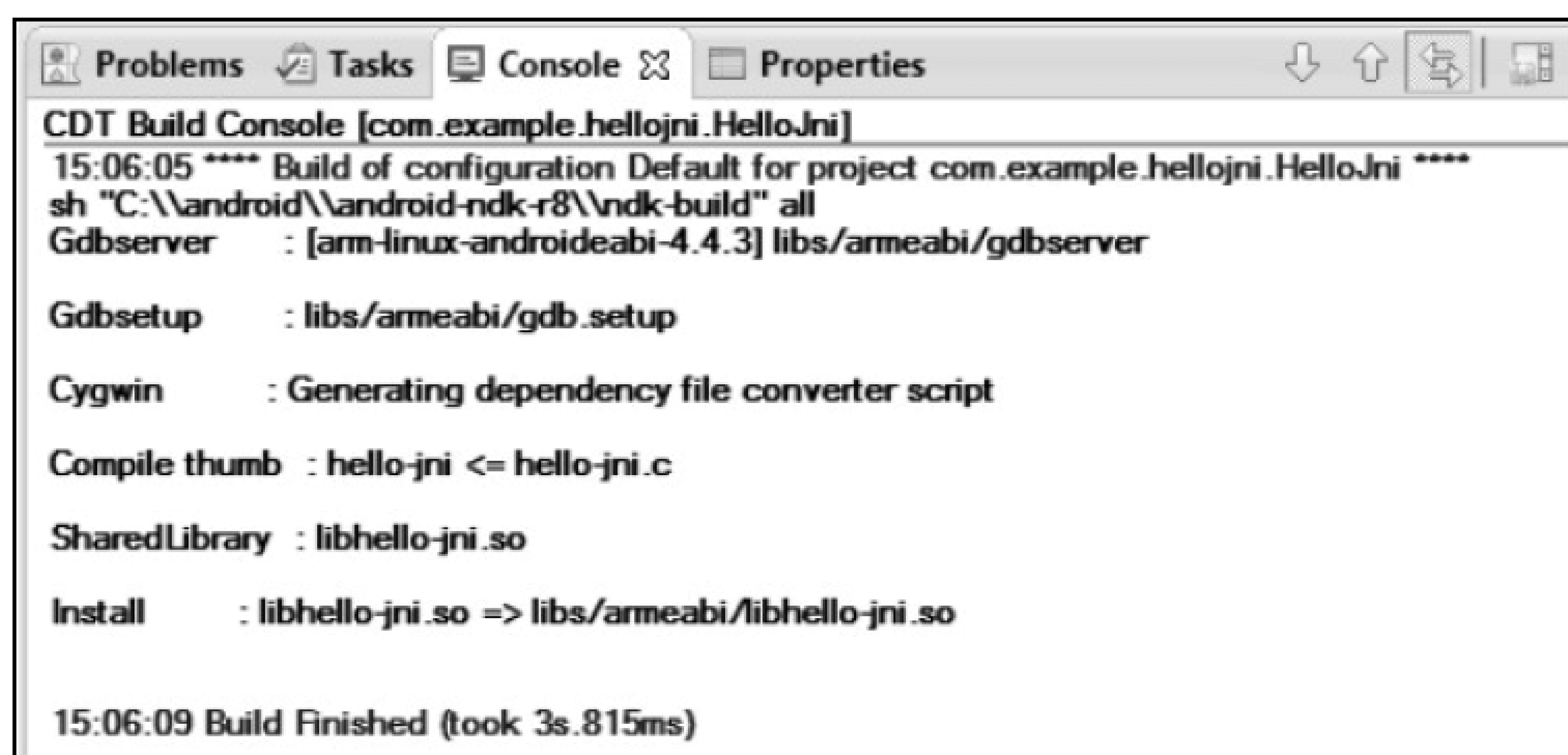


图 2-8 显示 Android NDK 构建信息的 Console 视图

尽管 Eclipse 可以很好地简化整个构建和部署过程，但就像本章之前所提到过的，Eclipse 不是构建 Android NDK 项目的必要条件，整个构建过程也可以用命令行方式执行。

2.3.5　用命令行对项目进行构建

为了在命令行方式下构建 hello-jni 项目，首先在 Windows 中打开命令提示符或在 Mac OS X 或 Linux 中打开终端窗口，并将 hello-jni project 所在目录更改为当前工作目录。用原生组件构建 Android 项目需要两步：第一步构建原生组件，第二步构建 Java 应用程序并将 Java 应用程序与其原生组件打包。为构建原生组件，在命令行方式下执行 ndk-build。ndk-build 是一个调用 Android 构建系统的辅助脚本。如图 2-9 所示，Android NDK 构建脚本会在构建过程中输出进度消息。



图 2-9　用 ndk-build 对原生组件进行构建

现在完成了原生组件的构建，可以继续第二步。Android SDK 构建系统是基于 Apache ANT 的。因为这是第一次用命令行构建项目，所以首先应该生成 Apache ANT 构建文件。在命令行中执行 android update project –p . –n hello-jni –t android-14--subprojects 命令来生成 Apache ANT 构建文件，如图 2-10 所示。

现在 Apache ANT 构建文档的编写已经完成，可以通过在命令行方式下执行“ant debug”命令构建项目，Apache ANT 将构建 Java 文件并将该 Java 文件与原生组件打成一个可安装 Android 包，即 APK 文件。通过上述操作可以看出，构建带有原生构件的 Android 应用最简单的方式是使用 Eclipse，因为不需要记住每一个构建操作步，所以不易出错。



图 2-10　生成 Apache ANT 构建文件

2.3.6 检测 Android NDK 项目的结构

现在重新回到 Eclipse 环境下学习带有原生组件的 Android 应用程序的结构。如图 2-11 所示，带有原生组件的 Android 项目包含一组附加的目录和文件。

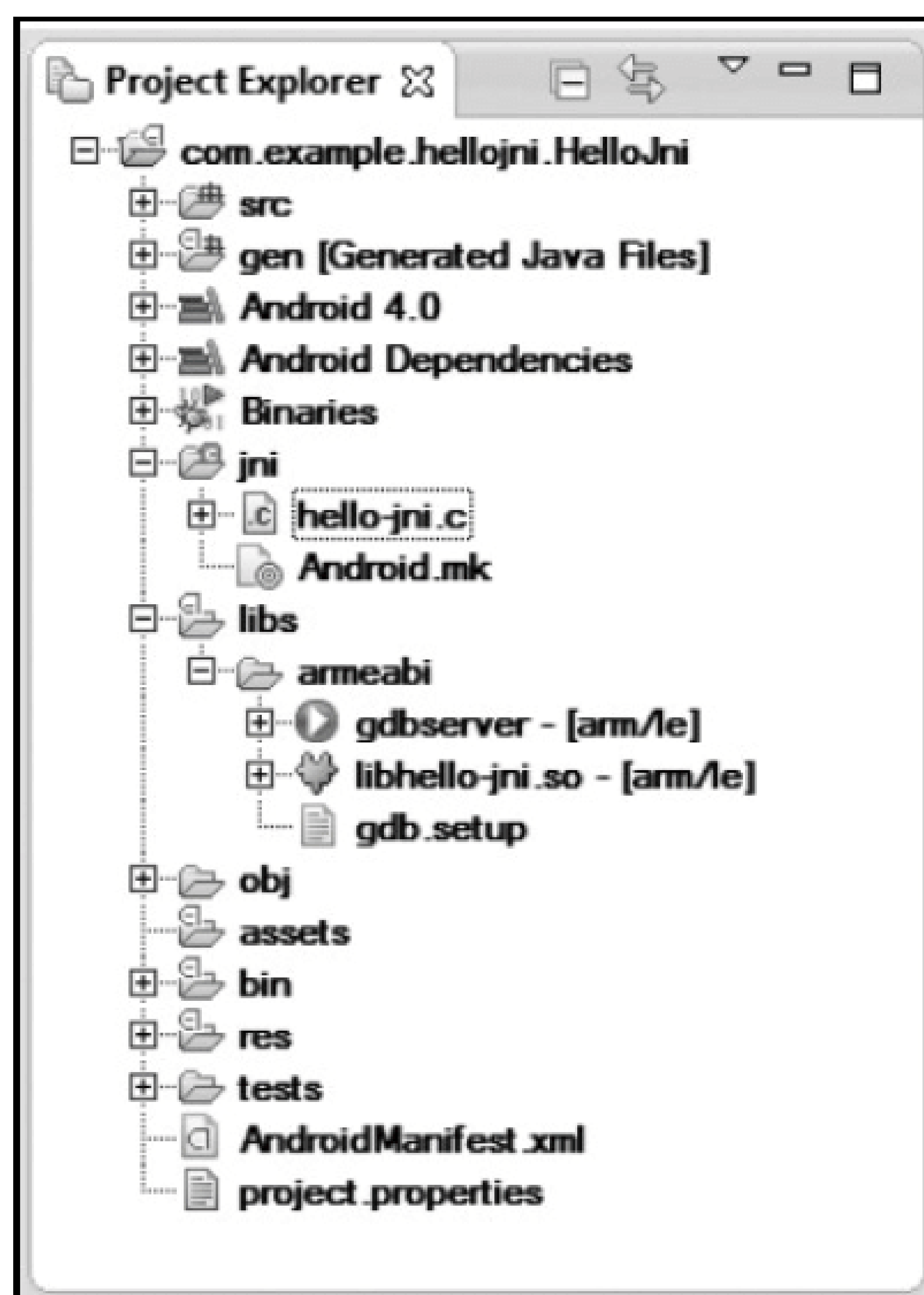


图 2-11　Hello-jni Android NDK 项目的结构

- **jni**：该目录包含原生组件的源代码以及描述原生组件构建方法的 Android.mk 构建文件。Android NDK 构建系统将该目录作为 NDK 项目目录并希望在项目根目录中找到它。
- **Libs**：在 Android NDK 构建系统的构建过程中创建该目录。它包含指定的目标机体系结构的独立子目录，例如 ARM 的 armeabi。在打包过程中该目录被包含在 APK 文件中。
- **Obj**：这是一个中间目录，编译源代码后所产生的目标文件都保存在该目录下。开发人员最好不要访问该目录。

Android NDK 项目最重要的组件是 Android.mk 构建文件，该文档描述了原生组件。理解构建系统是熟练运用 Android NDK 及其所有组件的关键。

2.4 构建系统

Android NDK 的构建系统是基于 GUN Make 的。该构建系统的主要目的是使开发人员能够用很短的构建文档来描述原生的 Android 应用程序；该构建系统还处理了包括替开发人员指定工具链、平台、CPU 和 ABI 等很多细节。封装该构建过程可以在不改变构建文件的情况下，使 Android NDK 的后续更新添加更多对工具链、平台以及系统接口的支持。

Android NDK 构建系统是由多种 GUN Makefile 片段构成的。该构建系统包括基于渲

染构建过程的不同类型 NDK 项目所需要的必要片段。如图 2-12 所示，这些构建系统片段可以在 Android NDK 安装程序的 build/core 子目录中找到。虽然开发人员并不会直接接触到这些文件，但知道它们的位置对与构建系统相关的故障很有帮助。

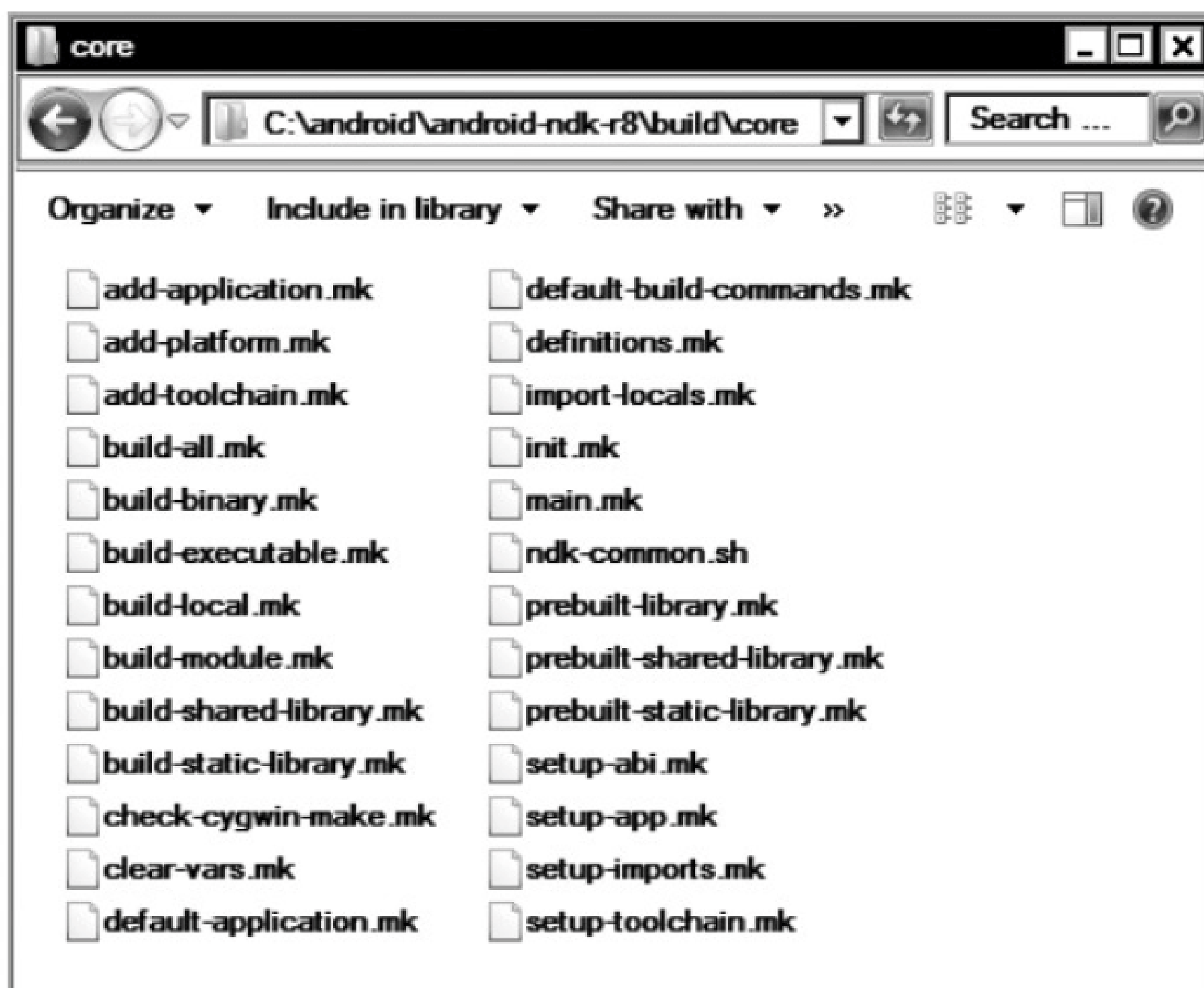


图 2-12　Android NDK 构建系统片段

除了这些片段，Android NDK 构建系统还要依赖另外两个文件：Android.mk 和 Application.mk，这两个文件应该作为 NDK 项目的一部分由开发人员提供，让我们来回顾一下。

2.4.1　Android.mk

Android.mk 是一个向 Android NDK 构建系统描述 NDK 项目的 GUN Makefile 片段。它是每一个 NDK 项目的必备组件。构建系统希望它出现在 jni 子目录中。在 Eclipse 的 Project Explorer 中，双击 Android.mk 文件在编辑视图中打开它。程序清单 2-1 显示了 hello-jni 项目中 Android.mk 文件的内容。

程序清单 2-1　hello-jni 项目下 Android.mk 文件的内容

```
# Copyright (C) 2009 The Android Open Source Project
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
```

```
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := hello-jni
LOCAL_SRC_FILES := hello-jni.c

include $(BUILD_SHARED_LIBRARY)
```

为了更好地理解它的句法，我们逐行分析。因为这个是一个 GUN Makefile 片段，所以它的句法和其他 Makefile 是一样的。每行都包含一个单独的指令，以“#”开头的是注释行，GUN Make 工具不处理它们。根据命名规范，变量名要大写。

注释块后的第一条指令是用来定义 LOCAL_PATH 变量的。根据 Android 构建系统的要求，Android.mk 文档必须以 LOCAL_PATH 变量的定义开头。

```
LOCAL_PATH :=$(call my-dir)
```

Android 构建系统利用 LOCAL_PATH 来定位源文件。因为将该变量设置为硬编码值并不合适，所以 Android 构建系统提供了一个名为 my-dir 的宏功能。通过将该变量设置为 my-dir 宏功能的返回值，可以将其放在当前目录下。

Android 构建系统将 CLEAR_VARS 变量设置为 clear-vars.mk 片段的位置。包含 Makefile 片段可以清除除了 LOCAL_PATH 以外的 LOCAL_<name>变量，例如 LOCAL_MODULE 与 LOCAL_SRC_FILES 等。

```
Include $(CLEAR_VARS)
```

这样做是因为 Android 构建系统在单次执行中解析多个构建文件和模块定义，而 LOCAL_<name>是全局变量。清除它们可以避免冲突，每一个原生组件被称为一个模块。

LOCAL_MODULE 变量用来给这些模块设定一个唯一的名称。下面的代码将该模块的名称设为 hello-jni：

```
LOCAL_MODULE  := hello-jni
```

因为模块名称也被用于给构建过程所生成的文件命名，所以构建系统给该文件添加了适当的前缀和后缀。本例中，hello-jni 模块会生成一个共享库文件且构建系统会将它命名为 libhello-jni.so。

用 LOCAL_SRC_FILES 变量定义用来建立和组装这个模块的源文件列表。

```
LOCAL_SRC_FILES := hello-jni.c
```

这里，hello-jni 模块只由一个源文件生成，而 LOCAL_SRC_FILES 变量可以包含用空

格分开的多个源文件名。

至此，Android.mk 文件中定义的构建系统变量简单描述了原生项目。编译和生成实际模块的构建系统还需要包含合适的构建系统片段，具体需要包含哪些片段取决于想要生成模块的类型。

1. 构建共享库

为了建立可供主应用程序使用的模块，必须将该模块变成共享库。Android NDK 构建系统将 BUILD_SHARED_LIBRARY 变量设置成 build-shared-library.mk 文件的保存位置。该 Makefile 片段包含了将源文件构建和组装成共享库的必要过程：

```
include  $(BUILD_SHARED_LIBRARY)
```

hello-jni 是一个简单的模块；然而，除非你的模块需要特殊处理，否则 Android.mk 文档将会包含一模一样的流程和指令。

2. 构建多个共享库

基于不同的应用程序的体系结构，一个单独的 Android.mk 文档可能产生多个共享库模块。为了达到这个目的，需要如程序清单 2-2 所示在 Android.mk 文档中定义多个模块。

程序清单 2-2　带有多个共享库模块的 Android.mk 构建文件

```
LOCAL_PATH := $(call my-dir)

#
#模块 1
#
include $(CLEAR_VARS)

LOCAL_MODULE := module1
LOCAL_SRC_FILES := module1.c
include $(BUILD_SHARED_LIBRARY)

#
#模块 2
#
include $(CLEAR_VARS)

LOCAL_MODULE := module2
LOCAL_SRC_FILES := module2.c

include $(BUILD_SHARED_LIBRARY)
```

在处理完这个 Android.mk 构建文档之后，Android NDK 构建系统会产生 libmodule1.so 和 libmodule2.so 两个共享库。

3. 构建静态库

Android NDK 构建系统也支持静态库。实际的 Android 应用程序并不直接使用静态库，并且应用程序包中也不包含静态库。静态库可以用来构建共享库。例如，在将第三方代码添加到现有原生项目中时，不用直接将第三方源代码包括在原生项目中，而是将第三方代码编译成静态库然后并入共享库，如程序清单 2-3 所示。

程序清单 2-3 在 Android.mk 文件中使用静态库

```
LOCAL_PATH := $(call my-dir)

#
# 第三方 AVI 库
#
include $(CLEAR_VARS)

LOCAL_MODULE := avilib
LOCAL_SRC_FILES := avilib.c platform_posix.c

include $(BUILD_STATIC_LIBRARY)

#
# 原生模块
#
include $(CLEAR_VARS)

LOCAL_MODULE := module
LOCAL_SRC_FILES := module.c

LOCAL_STATIC_LIBRARIES := avilib

include $(BUILD_SHARED_LIBRARY)
```

在将第三方代码模块生成静态库之后，共享库就可以通过将它的模块名添加到 LOCAL_STATIC_LIBRARIES 变量中来使用该模块。

4. 用共享库共享通用模块

静态库可以保证源代码模块化；但是，当静态库与共享库相连时，它就变成了共享库的一部分。在多个共享库的情况下，多个共享库与同一个静态库连接时，需要将通用模块的多个副本与不同共享库重复相连，这样就增加了应用程序的大小。在这种情况下，不用构建静态库，而是将通用模块作为共享库建立起来，而动态连接依赖模块以便消除重复的副本(见程序清单 2-4)。

程序清单 2-4 Android.mk 文件中共享库之间的代码共享

```
LOCAL_PATH := $(call my-dir)
```

```
#
# 第三方 AVI 库
#
include $(CLEAR_VARS)

LOCAL_MODULE := avilib
LOCAL_SRC_FILES := avilib.c platform_posix.c

include $(BUILD_SHARED_LIBRARY)

#
# 原生模块 1
#
include $(CLEAR_VARS)

LOCAL_MODULE := module1
LOCAL_SRC_FILES := module1.c

LOCAL_SHARED_LIBRARIES := avilib

include $(BUILD_SHARED_LIBRARY)

#
# 原生模块 2
#
include $(CLEAR_VARS)

LOCAL_MODULE := module2
LOCAL_SRC_FILES := module2.c

LOCAL_SHARED_LIBRARIES := avilib

include $(BUILD_SHARED_LIBRARY)
```

5. 在多个 NDK 项目间共享模块

同时使用静态库和共享库时，可以在模块间共享通用模块。但要说明的是，所有这些模块必须属于同一个 NDK 项目。从 R5 版本开始，Android NDK 也允许在 NDK 项目间共享和重用模块。考虑前面讲过的示例，可以通过以下步骤在多个 NDK 项目间共享 avilib 模块：

- 首先，将 avilib 源代码移动到 NDK 项目以外的位置，例如：C:\android\shared-modules\avilib。为了避免命名冲突，目录结构也可以包含模块提供者的名字，例如：C:\android\shared-modules \transcode\avilib。

注意：
在 Android NDK 构建系统中，共享模块路径不能包含空格。

- 作为共享模块，avilib 需要自己的 Android.mk 文件，如程序清单 2-5 所示。

程序清单 2-5　共享 avilib 模块的 Android.mk 文件

```
LOCAL_PATH := $(call my-dir)

#
#第三方 AVI 库
#
include $(CLEAR_VARS)

LOCAL_MODULE := avilib
LOCAL_SRC_FILES := avilib.c platform_posix.c

include $(BUILD_SHARED_LIBRARY)
```

- 现在，可以将 avilib 模块从 NDK 项目的 Android.mk 文件中移除。为了使用这个共享模块，将以 transcode/avilib 为参数调用函数宏 import-module 部分添加在构建文档的末尾，如程序清单 2-6 所示。为了避免构建系统的冲突，应该将 import-module 函数宏调用放在 Android.mk 文档的末尾。

程序清单 2-6　使用共享模块的 NDK 项目

```
#
# 原生模块
#
include $(CLEAR_VARS)

LOCAL_MODULE := module
LOCAL_SRC_FILES := module.c
LOCAL_SHARED_LIBRARIES := avilib

include $(BUILD_SHARED_LIBRARY)

$(call import-module,transcode/avilib)
```

- import-module 函数宏需要先定位共享模块，然后再将它导入到 NDK 项目中。默认情况下，import-module 函数宏只搜索<Android NDK>/sources 目录。为了搜索 c:\android\shared-modules 目录，定义一个名为 NDK_MODULE_PATH 的新环境变量并将它设置成共享模块的根目录，例如：c:\android\shared-modules。

6. 用 Prebuilt 库

使用共享模块要求有共享模块的源代码，Android NDK 构建系统简单地把这些源文件包含在 NDK 项目中并每次构建它们。自 R5 版本以后，Android NDK 也提供对 Prebuilt 库的支持。在下面的情况下，Prebuilt 库是非常有用的：

- 想在不发布源代码的情况下将你的模块发布给他人。
- 想使用共享模块的预建版来加速构建过程。

尽管已经被编译了，但预建模块仍需要一个 Android.mk 构建文档，如程序清单 2-7 所示。

程序清单 2-7　预构建共享模块的 Android.mk 文件

```
LOCAL_PATH := $(call my-dir)

#
# 第三方预构建 AVI 库
#
include $(CLEAR_VARS)

LOCAL_MODULE := avilib
LOCAL_SRC_FILES := libavilib.so

include $(PREBUILT_SHARED_LIBRARY)
```

LOCAL_SRC_FILES 变量指向的不是源文件，而是实际 Prebuilt 库相对于 LOCAL_PATH 的位置。

注意：

Prebuilt 库定义中不包含任何关于该库所构建的实际机器体系结构的信息。开发人员需要确保 Prebuilt 库是为与 NDK 项目相同的机器体系结构而构建的。

PREBUILT_SHARED_LIBRARY 变量指向 prebuilt-shared-library.mk Makefile 片段。它什么都没有构建，但是它将 Prebuilt 库复制到了 NDK 项目的 libs 目录下。通过使用 PREBUILT_STATIC_LIBRARY 变量，静态库可以像共享库一样被用作 Prebuilt 库，NDK 项目可以像普通共享库一样使用 Prebuilt 库了。

```
...
LOCAL_SHARED_LIBRARIES :=avilib
…
```

7. 构建独立的可执行文件

在 Android 平台上使用原生组件的推荐和支持的方法是将它们打包成共享库。但是，为了方便测试和进行快速原型设计，Android NDK 也支持构建独立的可执行文件。这些独立的可执行文件是不用打包成 APK 文件就可以复制到 Android 设备上的常规 Linux 应用程序，而且它们可以直接执行，而不通过 Java 应用程序加载。生成独立可执行文件需要在 Android.mk 构建文档中导入 BUILD_EXECUTABLE 变量，而不是导入 BUILD_SHARED_LIBRARY 变量，如程序清单 2-8 所示。

程序清单 2-8　独立可执行模块的 Android.mk 文件

```
#
# 独立的可执行的原生模块
#
include $(CLEAR_VARS)
```

```
LOCAL_MODULE := module
LOCAL_SRC_FILES := module.c

LOCAL_STATIC_LIBRARIES := avilib

include $(BUILD_EXECUTABLE)
```

BUILD_EXECUTABLE 变量指向 build-executable.mk Makefile 片段，该片段包含了在 Android 平台上生成独立可执行文件的必要步骤。独立可执行文件以与模块相同的名称被放在 libs/<machine architecture>目录下。尽管放在该目录下，但在打包阶段它并没有被包含在 APK 文件中。

8. 其他构建系统变量

除了在前几节提到的变量之外，Android NDK 构建系统还支持其他变量，本节将对这些变量进行简要说明。

构建系统定义的变量有：

- **TARGET_ARCH**：目标 CPU 体系结构的名称，例如 arm
- **TARGET_PLATFORM**：目标 Android 平台的名称，例如：android-3
- **TARGET_ARCH_ABI**：目标 CPU 体系结构和 ABI 的名称，例如：armeabi-v7a
- **TARGET_ABI**：目标平台和 ABI 的串联，例如：android-3-armeabi-v7a

可被定义为模块说明部分的变量有：

- **LOCAL_MODULE_FILENAME**：可选变量，用来重新定义生成的输出文件名称。默认情况下，构建系统使用 LOCAL_MODULE 的值作为生成的输出文件名称，但变量 LOCAL_MODULE_ FILENAME 可以覆盖 LOCAL_MODULE 的值。
- **LOCAL_CPP_EXTENSION**：C++源文件的默认扩展名是.cpp。这个变量可以用来为 C++源代码指定一个或多个文件扩展名。

```
…
LOCAL_CPP_ EXTENSION :=.cpp  .cxx
…
```

- **LOCAL_CPP_FEATURES**：可选变量，用来指明模块所依赖的具体 C++特性，如 RTTI、exceptions 等。

```
…
LOCAL_CPP_FEATURES :=rtti
…
```

- **LOCAL_C_INCLUDES**：可选目录列表，NDK 安装目录的相对路径，用来搜索头文件。

```
…
LOCAL_C_INCLUDES :=sources/shared-module
LOCAL_C_INCLUDES :=$(LOCAL_PATH)/include
…
```

- **LOCAL_CFLAGS：**一组可选的编译器标志，在编译 C 和 C++源文件的时候会被传送给编译器。

```
…
LOCAL_CFLAGS :=-DNDEBUG -DPORT=1234
…
```

- **LOCAL_CPP_FLAGS：**一组可选的编译标志，在只编译 C++源文件时被传送给编译器。
- **LOCAL_WHOLE_STATIC_LIBRARIES：**LOCAL_STATIC_LIBRARIES 的变体，用来指明应该被包含在生成的共享库中的所有静态库内容。

小贴士：
当几个静态库之间有循环依赖时，LOCAL_WHOLE_STATIC_LIBRARIES 很有用。

- **LOCAL_LDLIBS：**链接标志的可选列表，当对目标文件进行链接以生成输出文件时该标志将被传送给链接器。它主要用于传送要进行动态链接的系统库列表。例如：要与 Android NDK 日志库链接，使用以下代码：

```
LOCAL_LDFLAGS :=-llog
```

- **LOCAL_ALLOW_UNDEFINED_SYMBOLS：**可选参数，它禁止在生成的文件中进行缺失符号检查。若没有定义，链接器会在符号缺失时生成错误信息。
- **LOCAL_ARM_MODE：**可选参数，ARM 机器体系结构特有变量，用于指定要生成的 ARM 二进制类型。默认情况下，构建系统在拇指模式下用 16 位指令生成，但该变量可以被设置为 arm 来指定使用 32 位指令。

```
LOCAL_ARM_MODE :=arm
```

  该变量改变了整个模块的构建系统行为；可以用.arm 扩展名指定只在 arm 模式下构建特定文件。

```
LOCAL_SRC_FILES :=file1.c  file2.c.arm
```

- **LOCAL_ARM_NEON：**可选参数，ARM 机器体系结构特有变量，用来指定在源文件中应该使用的 ARM 高级单指令流多数据流(Single Instruction Multiple Data，SIMD)(a.k.a. NEON)内联函数。

```
LOCAL_ARM_NEON :=true
```

  该变量改变了整个模块的构建系统行为；可以用.neon 扩展名指定只构建带有 NEON 内联函数的特定文件。

```
LOCAL_SRC_FILES : =file1.c  file2.c.neon
```

- **LOCAL_DISABLE_NO_EXECUTE：**可选变量，用来禁用 NX Bit 安全特性。NX Bit 代表 Never Execute(永不执行)，它是在 CPU 中使用的一项技术，用来隔离代码

区和存储区。这样可以防止恶意软件通过将它的代码插入应用程序的存储区来控制应用程序。

```
LOCAL_DISABLE_NO_EXECUTE :=true
```

- **LOCAL_EXPORT_CFLAGS：**该变量记录一组编译器标志，这些编译器标志会被添加到通过变量 LOCAL_STATIC_LIBRARIES 或 LOCAL_SHARED_LIBRARIES 使用本模块的其他模块的 LOCAL_CFLAGS 定义中。

```
LOCAL_MODULE := avilib
...
LOCAL_EXPORT_CFLAGS := – DENABLE_AUDIO
...
LOCAL_MODULE := module1
LOCAL_CFLAGS := – DDEBUG
...
LOCAL_SHARED_LIBRARIES := avilib
```

  编译器在构建 module1 时会以-DENABLE_AUDIO –DDEBUG 标志执行。

- **LOCAL_EXPORT_CPPFLAGS：**和 LOCAL_EXPORT_CLAGS 一样，但是它是 C++特定代码编译器标志。
- **LOCAL_EXPORT_LDFLAGS：**和 LOCAL_EXPORT_CFLAGS 一样，但用作链接器标志。
- **LOCAL_EXPORT_C_INCLUDES：**该变量允许记录路径集，这些路径会被添加到通过变量 LOCAL_STATIC_LIBRARIES 或 LOCAL_SHARED_LIBRARIES 使用该模块的 LOCAL_C_INCLUDES 定义中。
- **LOCAL_SHORT_COMMANDS：**对于有大量资源或独立的静态/共享库的模块，该变量应该被设置为 true。诸如 Windows 之类的操作系统只允许命令行最多输入 8 191 个字符；该变量通过分解构建命令使其长度小于 8 191 个字符。在较小的模块中不推荐使用该方法，因为使用它会让构建过程变慢。
- **LOCAL_FILTER_ASM：**该变量定义了用于过滤来自 LOCAL_SRC_FILES 变量的装配文件的应用程序。

9. 其他的构建系统函数宏

本节概括了 Android NDK 构建系统支持的其他函数宏。

- **all-subdir-makefiles：**返回当前目录的所有子目录下的 Android.mk 构建文件列表。例如，调用以下命令可以将子目录下的所有 Android.mk 文件包含在构建过程中：

```
include $(call all-subdir-makefiles)
```

- **this-makefile：**返回当前 Android.mk 构建文件的路径。
- **parent-makefile：**返回包含当前构建文件的父 Android.mk 构建文件的路径。
- **grand-parent-makefile：**和 parent-makefile 一样但用于祖父目录。

10. 定义新变量

开发人员可以定义其他变量来简化他们的构建文件。以 LOCAL_和 NDK_前缀开头的名称预留给 Android NDK 构建系统使用。建议开发人员定义的变量以 MY_开头，如程序清单 2-9 所示。

程序清单 2-9　表示开发人员定义的中间变量的使用方法的 Android.mk 文件

```
...
MY_SRC_FILES := avilib.c platform_posix.c
LOCAL_SRC_FILES := $(addprefix avilib/, $(MY_SRC_FILES))
...
```

11. 条件操作

Android.mk 构建文件也可以包含关于这些变量的条件操作，例如：在每个体系结构中包含一个不同的源文件集，如程序清单 2-10 所示。

程序清单 2-10　包含条件操作的构建文件 Android.mk

```
...
ifeq ($(TARGET_ARCH),arm)
    LOCAL_SRC_FILES + = armonly.c
else
    LOCAL_SRC_FILES + = generic.c
endif
...
```

2.4.2　Application.mk

Application.mk 是 Android NDK 构建系统使用的一个可选构建文件。和 Android.mk 文件一样，它也被放在 jni 目录下。Application.mk 也是一个 GUN Makefile 片段。它的目的是描述应用程序需要哪些模块；它也定义所有模块的通用变量。以下是 Application.mk 构建文件支持的变量：

- **APP_MODULES：**默认情况下，Android NDK 构建系统构建 Android.mk 文件声明的所有模块。该变量可以覆盖上述行为并提供一个用空格分开的、需要被构建的模块列表。
- **APP_OPTIM：**该变量可以被设置为 release 或 debug 以改变生成的二进制文件的优化级别。默认情况下使用的是 release 模式，并且此时生成的二进制文件被高度优化。该变量可以被设置为 debug 模式以生成更容易调试的未优化二进制文件。
- **APP_CLAGS：**该变量列出了一些编译器标志，在编译任何模块的 C 和 C++源文件时这些标志都会被传给编译器。
- **APP_CPPFLAGS：**该变量列出了一些编译器标志，在编译任何模块的 C++源文件时这些标志都会被传给编译器。

- **APP_BUILD_SCRIPT：** 默认情况下，Android NDK 构建系统在项目的 jni 子目录下查找 Android.mk 构建文件。可以用该变量改变上述行为，并使用不同的生成文件。
- **APP_ABI：** 默认情况下，Android NDK 构建系统为 armeabi ABI 生成二进制文件。可以用该变量改变上述行为，并为其他 ABI 生成二进制文件，例如：

  ```
  APP_ABI := mips
  ```

  另外，可以设置多个 ABI

  ```
  APP_ABI := armeabi mips
  ```

  为所有支持的 ABI 生成二进制文件

  ```
  APP_ABI := all
  ```

- **APP_STL：** 默认情况下，Android NDK 构建系统使用最小 STL 运行库，也被称为 system 库。可以用该变量选择不同的 STL 实现。

  ```
  APP_STL :=stlport_shared
  ```

- **APP_GNUSTL_FORCE_CPP_FEATURES：** 与 LOCAL_CPP_EXTENSIONS 变量相似，该变量表明所有模块都依赖于具体的 C++特性，如 RTTI、exceptions 等。
- **APP_SHORT_COMMANDS：** 与 LOCAL_SHORT_COMMANDS 变量相似，该变量使得构建系统在有大量源文件的情况下可以在项目中使用更短的命令。

2.5 使用 NDK-Build 脚本

如前所述，可以通过执行 ndk-build 脚本启动 Android NDK 构建系统。该脚本用一组参数使维护和控制构建过程更容易。

- 默认情况下，ndk-build 脚本应该在主项目目录中执行。-C 参数可以用于指定命令行中 NDK 项目的位置，这样一来 ndk-build 脚本可以从任意的位置开始。

  ```
  ndk-build  -C  /path/to/the/project
  ```

- 如果源文件没被修改，Android NDK 构建系统不会重构建目标。可以用-B 执行 ndk-build 脚本来强制重构建所有源代码。

  ```
  ndk-build  -B
  ```

- 为了清理生成的二进制文件和目标文件，可以在命令行执行 ndk-build clean 命令。Android NDK 构建系统会删除生成的二进制文件。

  ```
  ndk-build clean
  ```

- Android NDK 构建系统依赖于 GNU Make 工具对模块进行构建。默认情况下，GNU Make 工具一次执行一句构建命令，等这一句完成以后再执行下一句。如果在命令

行使用-j 参数，GNU Make 就可以并行执行构建命令。另外，也可以通过指定该参数之后的数字来指定并行执行的命令总数。

```
ndk-build  -j  4
```

2.6　排除构建系统故障

Android NDK 构建系统有大量的日志以支持构建系统相关的故障排除，本节将简要阐述构建系统故障排除。

在命令行输入 ndk-build　NDK_LOG=1 便可启用 Android NDK 构建系统内部状态日志功能。Android NDK 构建系统会产生大量的日志，日志消息的前缀是 Android NDK(如图 2-13 所示)。

```
C:\windows\system32\cmd.exe - ndk-build  NDK_LOG=1

C:\android\workspace\com.example.hellojni.HelloJni>ndk-build NDK_LOG=1
Android NDK: NDK installation path auto-detected: 'C:/android/android-ndk-r8'
Android NDK: GNU Make version 3.81 detected
Android NDK: Host OS was auto-detected: windows
Android NDK:  Host operating system detected: windows
Android NDK: Host CPU was auto-detected: x86
Android NDK: HOST_TAG set to windows
Android NDK: Host tools prebuilt directory: C:/android/android-ndk-r8/prebuilt/w
indows/bin
Android NDK: Host 'echo' tool: C:/android/android-ndk-r8/prebuilt/windows/bin/ec
ho.exe
Android NDK: Host 'awk' tool: C:/android/android-ndk-r8/prebuilt/windows/bin/awk
.exe
Android NDK: Host 'awk' test returned: Pass
Android NDK:  This NDK supports the following target architectures and ABIS:
Android NDK:    arm: armeabi armeabi-v7a
Android NDK:    mips: mips
Android NDK:    x86: x86
```

图 2-13　ndk-build 脚本显示调试信息

如果只想看实际执行的构建命令，可以在命令行输入 ndk-build V=1。Android NDK 将会只显示构建命令，如图 2-14 所示。

```
C:\windows\system32\cmd.exe
C:\android\workspace\com.example.hellojni.HelloJni>ndk-build V=1
Gdbserver      : [arm-linux-androideabi-4.4.3] libs/armeabi/gdbserver
if not exist ".\libs\armeabi" md ".\libs\armeabi"
copy /b/y "C:\android\android-ndk-r8\toolchains\arm-linux-androideabi-4.4.3\preb
uilt\gdbserver" ".\libs\armeabi\gdbserver" > NUL
Gdbsetup       : libs/armeabi/gdb.setup
if not exist ".\libs\armeabi" md ".\libs\armeabi"
C:/android/android-ndk-r8/prebuilt/windows/bin/echo.exe "set solib-search-path .
/obj/local/armeabi" > ./libs/armeabi/gdb.setup
C:/android/android-ndk-r8/prebuilt/windows/bin/echo.exe "directory C:/android/an
droid-ndk-r8/platforms/android-14/arch-arm/usr/include jni C:/android/android-nd
k-r8/sources/cxx-stl/system" >> ./libs/armeabi/gdb.setup
Install        : hello-jni => libs/armeabi/hello-jni
if not exist ".\libs\armeabi" md ".\libs\armeabi"
copy /b/y ".\obj\local\armeabi\hello-jni" ".\libs\armeabi\hello-jni" > NUL
C:/android/android-ndk-r8/toolchains/arm-linux-androideabi-4.4.3/prebuilt/window
s/bin/arm-linux-androideabi-strip --strip-unneeded  ./libs/armeabi/hello-jni

C:\android\workspace\com.example.hellojni.HelloJni>
```

图 2-14　ndk-build 脚本显示构建命令

2.7 小结

本章引导读者学习了在 Eclipse IDE 环境下构建 NDK 项目的方法，在此过程中讲述了很多 Android NDK 构建系统的知识，第 3 章将继续学习用 Android NDK 构建的原生组件如何与实际的 Java 应用程序通信。

第 3 章

用 JNI 实现与原生代码通信

第 2 章通过学习 Android NDK 的组件、结构和构建系统对 Android NDK 有了初步的了解，现在可以构建任何一段原生代码并将其与 Android 应用一起打包。本章将集中讨论使用 Java 原生接口(JNI，Java Native Interface)技术实现 Java 应用程序和原生代码之间通信。

3.1 什么是 JNI

JNI 是 Java 程序设计语言功能最强的特征，它允许 Java 类的某些方法原生实现，同时让它们能够像普通 Java 方法一样被调用和使用。这些原生方法也可以使用 Java 对象，使用方法与 Java 代码使用 Java 对象的方法相同。原生方法可以创建新的 Java 对象或者使用 Java 应用程序创建的对象，这些 Java 应用程序可以检查、修改和调用这些对象的方法以执行任务。

3.2 以一个示例开始

在详细讲解 JNI 技术之前，我们先看一个示例应用程序。这会为本章的概念学习和 API 实验提供必要的基础。学习了这个示例应用程序之后，我们将会掌握以下主要概念：

- Java 代码如何调用原生方法
- 声明原生方法
- 在共享库中载入原生模块
- 在 C/C++中实现原生方法

首先，打开 Eclipse 集成开发环境，进入第 2 章引入的 hello-jni 示例项目。hello-jni 应用程序是单个活跃的 Android 应用程序。在 Project Explorer 视图中展开 src 目录，再展开 com.example.hellojni 包。双击 HelloJni.java 源文件在 Editor 视图中打开 HelloJni activity。

helloJni activity 有一个由单个 android.widget.TextView widget 构成的很简单的用户接

口。在 activity 的 onCreate 方法体中，TextView widget 的字符串值被设置成 stringFromJNI 方法的返回值，如程序清单 3-1 所示。

程序清单 3-1　HelloJni Activity onCreate Method

```
/** 当activity 首次创建时调用. */
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);

    /* 创建一个文本视图并设置其内容
     * 通过调用一个原生函数检索文本
     */
    TextView tv = new TextView(this);
    tv.setText( stringFromJNI() );
    setContentView(tv);
}
```

这里没有什么新内容，在 onCreate 方法的下面可以看到 stringFromJNI 方法。

3.2.1　原生方法的声明

如程序清单 3-2 所示，stringFromJNI 方法声明中含有关键字 native 以通知 Java 编译器，它用另一种语言提供该方法的具体实现。因为原生方法没有方法体，方法声明以语句终结符——分号结尾。

程序清单 3-2　原生 stringFromJNI 方法的方法声明

```
/* 原生方法由'hello-jni'原生库实现
 * 'hello-jni'，该原生库与本应用程序一起打包
 */
public native String stringFromJNI();
```

尽管现在虚拟机知道该方法被原生实现，但是它仍然不知道到哪儿去找方法的实现。

3.2.2　加载共享库

第 2 章提到，原生方法被编译成一个共享库。需要先加载该共享库以便于虚拟机能够找到原生方法实现。java.lang.System 类提供了两个静态方法，load 和 loadLibrary，用于在运行时加载共享库。如程序清单 3-3 所示，HelloJni activity 加载 hello-jni 共享库。

程序清单 3-3　HelloJni Activity 加载 hello-jni 共享库

```
/* 这段代码用于在应用启动时加载'hello-jni'库
 * 该库在安装时由包管理器
 * 解压到/data/data/com.example.HelloJni/lib/libhello-jni.so 中
 */
```

```
static {
        System.loadLibrary("hello-jni");
}
```

因为想加载原生代码实现，正如类首次加载和初始化一样，所以在静态上下文中调用 loadLibrary 方法。

请记住，Java 技术的设计目标是平台独立，作为 Java 框架 API 的一部分，loadLibrary 也要保持平台独立性。尽管 Android NDK 生成的实际共享库被命名为 libhello-jni.so，但是 loadLibrary 方法只采用 hello-jni 这个库名，再按照所使用的具体操作系统的要求加上必要的前缀或后缀。库名与 Android.mk 文件中使用 LOCAL_MODULE 构建系统变量定义的模块名相同。

loadLibrary 的参数也不包含共享库的位置。Java 库路径，也就是系统属性 java.library.path 保存 loadLibrary 方法在共享库搜索的目录列表，Android 上的 Java 库路径包含/vendor/lib 和/system/lib。

需要强调的是，loadLibrary 在扫描 Java 库路径时，一旦发现同名的库，立即加载共享库。因为 Java 库路径的第一组目录是 Android 系统目录，为了避免与系统库命名冲突，强烈建议 Android 开发人员为每个共享库选择唯一的名字。

现在我们看看原生代码，学习原生方法的声明和实现方法。

3.2.3　实现原生方法

在 Project Explorer 视图中，展开 jni 目录并双击 hello-jni.c 源文件在 Editor 视图中打开该文件。如程序清单 3-4 所示，C 源代码文件以 jni.h 头文件包含语句开头，这个头文件中包含 JNI 数据类型和函数的定义。

程序清单 3-4　stringFromJNI 方法的原生实现

```
#include <string.h>
#include <jni.h>
...
Jstring
Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env,
                                                  jobject thiz )
{
    return (*env)->NewStringUTF(env, "Hello from JNI !");
}
```

原生方法 stringFromJNI 也用一个名为 Java_com_example_hellojni_HelloJni_stringFromJNI 的完全限定的函数来声明，这种显式的函数命名让虚拟机在加载的共享库中自动查找原生函数。

1. C/C++头文件生成器：javah

让原生函数名及参数列表与 Java 类文件的原始定义一致是繁杂而多余的，因为 JDK 自带一个名为 javah 的命令行工具来执行任务，javah 工具可以为原生方法解析 Java 类文件

并生成由原生方法声明组成的头文件。

(1) 在命令行方式下运行

在命令行方式下，将当前工作目录改为 HelloJni 项目的导入目录，即<Eclipse Workspace>/com.example.hellojni.HelloJni。javah 工具对编译过的 Java 类文件进行操作，用编译过的类文件所在位置和要解析的 Java 类名为参数调用 javah，命令格式如下：

```
javah -classpath bin/classes com.example.hellojni.HelloJni
```

javah 工具将解析 com.example.hellojni.HelloJni 类文件，且生成名为 com_example_hellojni_HelloJni.h 的 C/C++头文件，文件内容如程序清单 3-5 所示。

程序清单 3-5　头文件 com_example_hellojni_HelloJni.h

```
/* 不要编辑这个文件-它是机器自动生成的*/
#include < jni.h>
/* com_example_hellojni_HelloJni 类的头文件*/

#ifndef _Included_com_example_hellojni_HelloJni
#define _Included_com_example_hellojni_HelloJni
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class: com_example_hellojni_HelloJni
 * Method: stringFromJNI
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_stringFromJNI
  (JNIEnv *, jobject);
/*
 * Class: com_example_hellojni_HelloJni
 * Method: unimplementedStringFromJNI
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_
unimplementedStringFromJNI
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

C/C++ 源文件只需要包含这个头文件并提供原生方法的实现，如程序清单 3-6 所示。

程序清单 3-6　com_example_hellojni_HelloJni.c 源文件

```
#include "com_example_hellojni_HelloJni.h"
```

```
JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_stringFromJNI
  (JNIEnv * env, jobject thiz)
{
    return (*env)->NewStringUTF(env, "Hello from JNI !");
}
```

并不需要每次都在命令行方式下运行 javah 工具，为了简化头文件生成的过程，它可以被集成到 Eclipse 中成为 Eclipse 的外部工具。

(2) 在 Eclipse IDE 中运行

打开 Eclipse IDE，在顶部菜单栏选择 Run | External Tools External Tools Configurations。在 External Tools Configurations 对话框中选择 Program，单击 New launch configuration 按钮，单击 Main 选项卡，如图 3-1 所示，按照下面的内容填写工具信息：

- **Name：** Generate C and C++ Header File
- **Location：** ${system_path:javah}
- **Working Directory：** ${project_loc}/jni
- **Arguments：** -classpath "${project_classpath};${env_var:ANDROID_SDK_HOME}/platforms/android-14/android.jar" ${java_type_name}

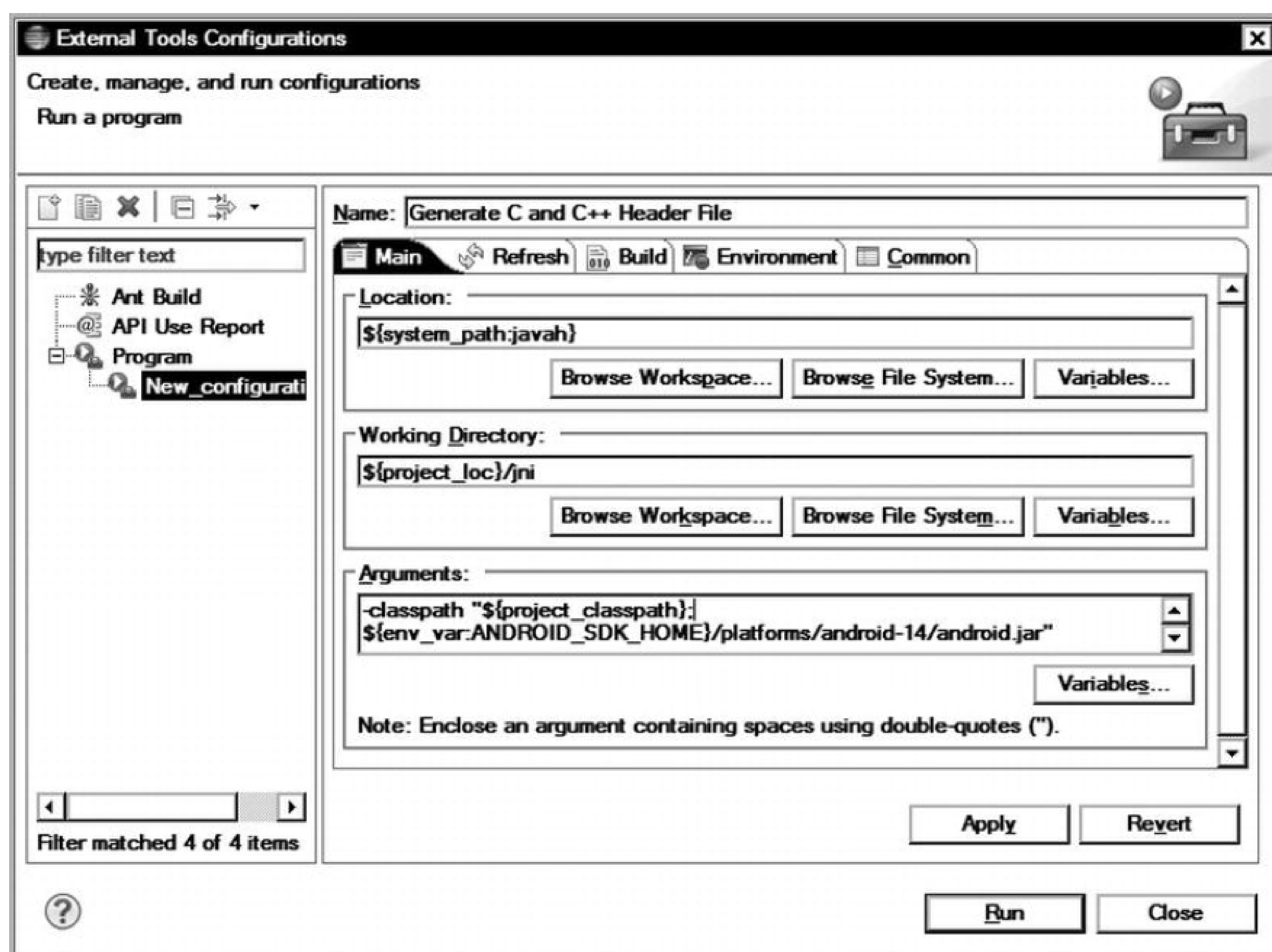


图 3-1　javah 外部工具配置

在 Mac OS X 和 Linux 平台上需要冒号代替分号。切换到 Refresh 选项卡，选中 Refresh resource upon completion 复选框，并在列表中选择 The project containing the selected resource，如图 3-2 所示。

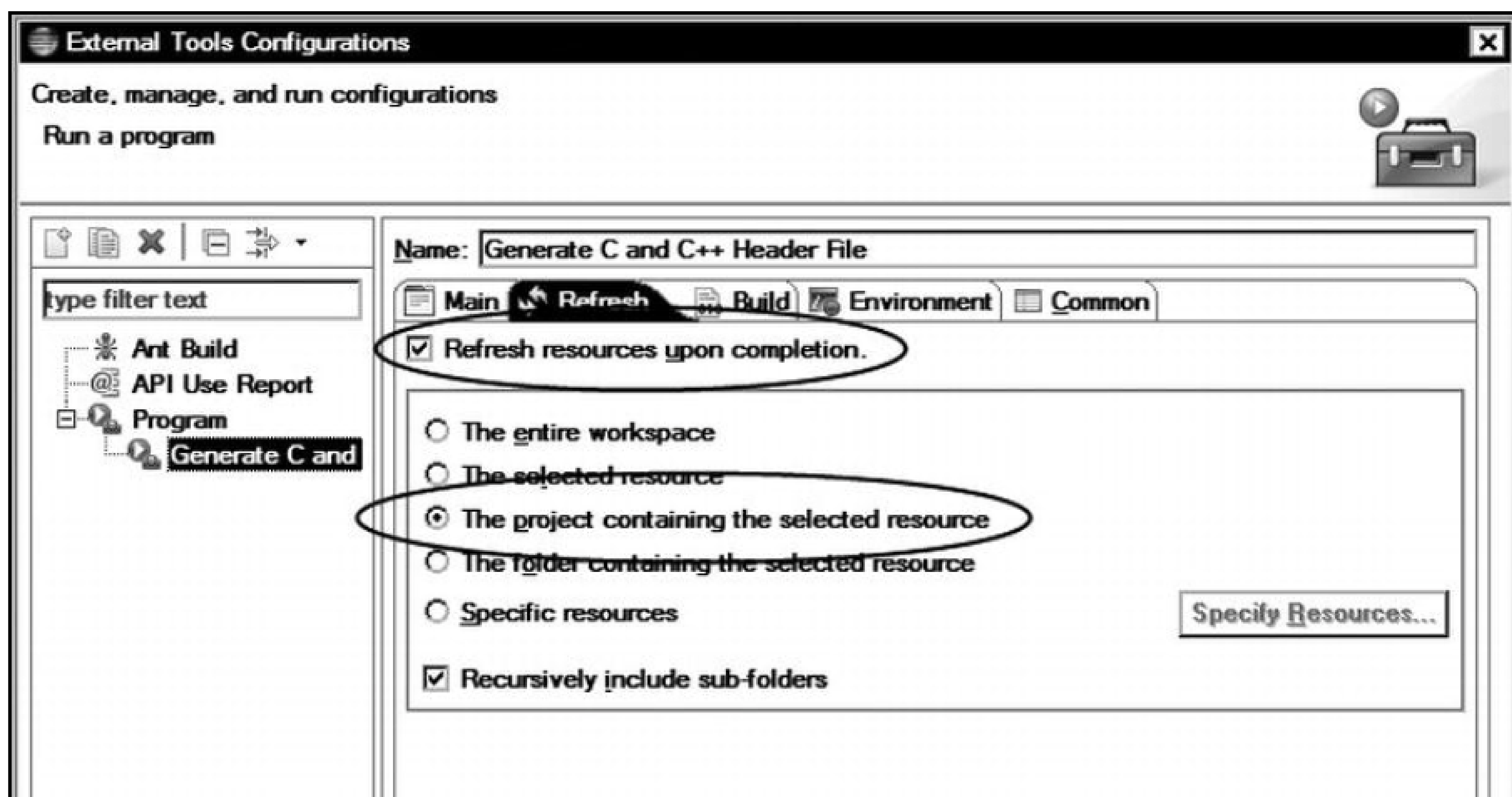


图 3-2　运行 javah 工具时刷新项目

切换到 Common 选项卡，选中 Display in favorites menu 组下面的复选框 External Tools，如图 3-3 所示。

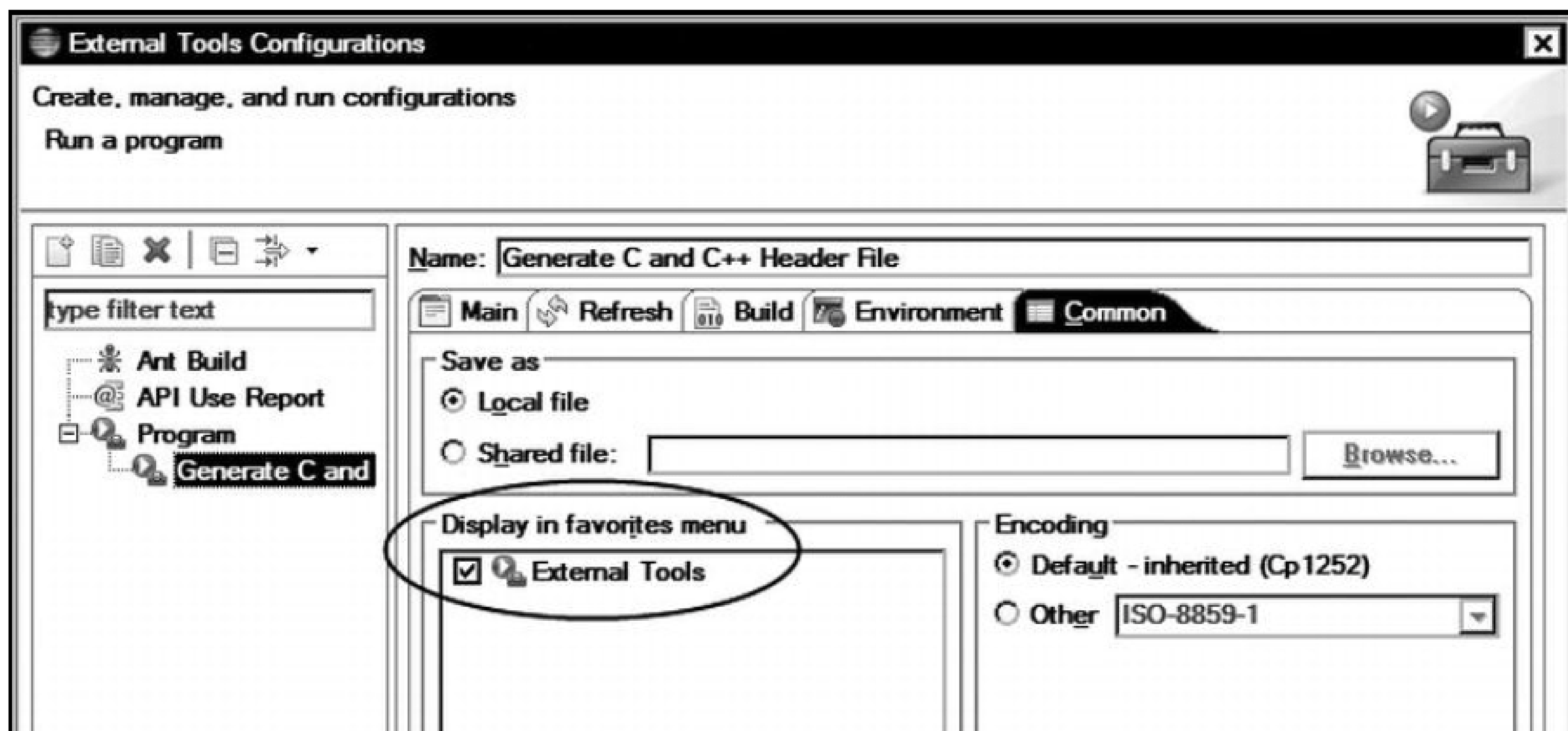


图 3-3　在收藏夹菜单中显示 javah 工具

单击 OK 按钮保存外部工具配置。为了测试配置的效果，在 Project Explorer 视图中选择 HelloJni 类，选择 Run | External Tools | Generate C and C++ Header File，javah 工具将为原生方法解析选中的类文件，并在 jni 目录下生成一个名为 com_example_hellojni_HelloJni.h 的、带有方法描述的 C/C++头文件。

既然你已经掌握了自动生成原生方法声明的方法，下面我们详细介绍生成的方法声明。

2. 方法声明

尽管 Java 方法 stringFromJNI 不带任何参数，但是原生方法带两个参数，如程序清单 3-7 所示。

程序清单 3-7　原生方法的强制参数

```
JNIEXPORT jstring JNICALL
Java_com_example_hellojni_HelloJni_stringFromJNI(JNIEnv *, jobject);
```

第一个参数 JNIEnv 是指向可用 JNI 函数表的接口指针；第二个参数 jobject 是 HelloJni 类实例的 Java 对象引用。

(1) JNIEnv 接口指针

原生代码通过 JNIEnv 接口指针提供的各种函数来使用虚拟机的功能。JNIEnv 是一个指向线程-局部数据的指针，而线程-局部数据中包含指向函数表的指针。实现原生方法的函数将 JNIEnv 接口指针作为它们的第一个参数。

注意

传递给每一个原生方法调用的 JNIEnv 接口指针在与方法调用相关的线程中也有效，但是它不能被缓存以及被其他线程使用。

原生代码是 C 与原生代码是 C++其调用 JNI 函数的语法不同。C 代码中，JNIEnv 是指向 JNINativeInterface 结构的指针，为了访问任何一个 JNI 函数，该指针需要首先被解引用。因为 C 代码中的 JNI 函数不了解当前的 JNI 环境，JNIEnv 实例应该作为第一个参数传递给每一个 JNI 函数调用调用者，调用格式如下：

```
return (*env)->NewStringUTF(env, "Hello from JNI !");
```

在 C++代码中，JNIEnv 实际上是 C++类实例，JNI 函数以成员函数的形式存在。因为 JNI 方法已经访问了当前的 JNI 环境，因此 JNI 方法调用不要求 JNIEnv 实例作参数。在 C++中，完成同样功能的调用代码格式如下：

```
return env->NewStringUTF("Hello from JNI !");
```

(2) 实例方法与静态方法

Java 程序设计语言有两类方法：实例方法和静态方法。实例方法与类实例相关，它们只能在类实例中调用。静态方法不与类实例相关，它们可以在静态上下文直接调用。静态方法和实例方法均可以声明为原生的，可以通过 JNI 技术以原生代码的形式提供它们的实现。原生实例方法通过第二个参数获取实例引用，该参数是 jobject 类型的，如程序清单 3-8 所示。

程序清单 3-8　原生实例方法定义

```
JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_stringFromJNI
(JNIEnv * env, jobject thiz);
```

因为静态方法没有与实例绑定，因此通过第二个参数获取类引用而不是实例引用，第二个参数是 jclass 值类型的，如程序清单 3-9 所示。

程序清单 3-9　原生静态方法定义

```
    JNIEXPORT jstring JNICALL
Java_com_example_hellojni_HelloJni_stringFromJNI(JNIEnv * env, jclass clazz);
```

正如在方法定义中看到的，JNI 提供了自己的数据类型从而让原生代码了解 Java 数据类型。

3.3　数据类型

Java 中有两种数据类型：

- 基本数据类型：布尔型、字节型、字符型、短整型、整型、长整型、浮点型和双精度类型。
- 引用类型：字符串类、数组类及其他类。

我们将进一步学习每种数据类型。

3.3.1　基本数据类型

基本数据类型可以直接与 C/C++的相应基本数据类型映射，如表 3-1 所示。JNI 用类型定义使得这种映射对开发人员透明。

表 3-1　Java 基本数据类型

Java 类型	JNI 类型	C/C++类型	大小
Boolean	Jblloean	unsigned char	无符号 8 位
Byte	Jbyte	char	有符号 8 位
Char	Jchar	unsigned short	无符号 16 位
Short	Jshort	short	有符号 16 位
Int	Jint	int	有符号 32 位
Long	Jlong	long long	有符号 64 位
Float	Jfloat	float	32 位
Double	Jdouble	double	64 位

3.3.2　引用类型

与基本数据类型不同，引用类型对原生方法是不透明的，引用类型映射如表 3-2 所示。它们的内部数据结构并不直接向原生代码公开。

表 3-2　Java 引用类型映射

Java 类型	原生类型
java.lang.Class	jclass
java.lang.Throwable	jthrowable
java.lang.String	jstring
Other objects	jobjects
java.lang.Object[]	jobjectArray
boolean[]	jbooleanArray
byte[]	jbyteArray
char[]	jcharArray
short[]	jshortArray
int[]	jintArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray
Other arrays	Jarray

3.4　对引用数据类型的操作

引用类型以不透明的引用方式传递给原生代码，而不是以原生数据类型的形式呈现，因此引用类型不能直接使用和修改。JNI 提供了与这些引用类型密切相关的一组 API，这些 API 通过 JNIEnv 接口指针提供给原生函数。本节将简单介绍与下列类型和组件相关的 API：

- 字符串
- 数组
- NIO 缓冲区
- 字段
- 方法

3.4.1　字符串操作

JNI 把 Java 字符串当成引用类型来处理。这些引用类型并不像原生 C 字符串一样可以直接使用，JNI 提供了 Java 字符串与 C 字符串之间相互转换的必要函数。因为 Java 字符串对象是不可变的，因此 JNI 不提供任何修改现有的 Java 字符串内容的函数。

JNI 支持 Unicode 编码格式和 UTF-8 编码格式的字符串，还提供两组函数通过 JNIEnv 接口指针处理这些字符串编码。

1. 创建字符串

可以在原生代码中用 NewString 函数构建 Unicode 编码格式的字符串实例，用 NewStringUTF 函数构建 UTF-8 编码格式的字符串实例。正如程序清单 3-10 所示，这些函数以一个 C 字符串为参数，并返回一个 Java 字符串引用类型 jstring 值。

程序清单 3-10　用给定的 C 字符串创建 Java 字符串

```
jstring javaString;
javaString = (*env)->NewStringUTF(env, "Hello World!");
```

在内存溢出的情况下，这些函数返回 NULL 以通知原生代码虚拟机中抛出异常，这样原生代码就会停止运行，本章的后面几节会介绍异常处理相关内容。

2. 把 Java 字符串转换成 C 字符串

为了在原生代码中使用 Java 字符串，需要先将 Java 字符串转换成 C 字符串。用 GetStringChars 函数可以将 Unicode 格式的 Java 字符串转换成 C 字符串，用 GetString-UTFChars 函数可以将 UTF-8 格式的 Java 字符串转换成 C 字符串。这些函数的第三个参数均为可选参数，该可选参数名是 isCopy，它让调用者确定返回的 C 字符串地址指向副本还是指向堆中的固定对象，如程序清单 3-11 所示。

程序清单 3-11　将 Java 字符串转换成 C 字符串

```
const jbyte* str;
jboolean isCopy;

str = (*env)->GetStringUTFChars(env, javaString, &isCopy);
if (0 != str) {
    printf("Java string: %s", str);

    if (JNI_TRUE == isCopy) {
        printf("C string is a copy of the Java string.");
    } else {
        printf("C string points to actual string.");
    }
}
```

3. 释放字符串

通过 JNI GetStringChars 函数和 GetStringUTFChars 函数获得的 C 字符串在原生代码中使用完之后需要正确地释放，否则将会引起内存泄露。如程序清单 3-12 所示，JNI 提供了 ReleaseStringChars 函数释放 Unicode 编码格式的字符串，而用 ReleaseStringUTFChars 函数释放 UTF-8 编码格式的字符串。

程序清单 3-12　释放 JNI 函数返回的 C 字符串

```
(*env)->ReleaseStringUTFChars(env, javaString, str);
```

3.4.2　数组操作

JNI 把 Java 数组当成引用类型来处理，JNI 提供必要的函数访问和处理 Java 数组。

1. 创建数组

用 New<*Type*>Array 函数在原生代码中创建数组实例，其中<Type>可以是 Int、Char 和 Boolean 等，例如 NewIntArray。如程序清单 3-13 所示，使用这些函数时应该以参数的形式给出数组大小。

程序清单 3-13　在原生代码中创建数组

```
jintArray javaArray;
javaArray = (*env)->NewIntArray(env, 10);
if (0 != javaArray) {
    /* 现在可以使用数组了. */
}
```

与 NewString 函数一样，在内存溢出的情况下，New<Type>Array 函数将返回 NULL 以通知原生代码虚拟机中有异常抛出，这样原生代码就会停止运行。

2. 访问数组元素

JNI 提供两种访问 Java 数组元素的方法，可以将数组的代码复制成 C 数组或者让 JNI 提供直接指向数组元素的指针。

3. 对副本的操作

Get<Type>ArrayRegion 函数将给定的基本 Java 数组复制到给定的 C 数组中，如程序清单 3-14 所示。

程序清单 3-14　将 Java 数组区复制到 C 数组中

```
jint nativeArray[10];
(*env)->GetIntArrayRegion(env, javaArray, 0, 10, nativeArray);
```

原生代码可以像使用普通的 C 数组一样使用和修改数组元素。当原生代码想将所做的修改提交给 Java 数组时，可以使用 Set<Type>ArrayRegion 函数将 C 数组复制回 Java 数组中，如程序清单 3-15 所示。

程序清单 3-15　从 C 数组向 Java 数组提交所作的修改

```
(*env)->SetIntArrayRegion(env, javaArray, 0, 10, nativeArray);
```

当数组很大时，为了对数组进行操作而复制数组会引起性能问题。在这种情况下，如果可能的话，原生代码应该只获取或设置数组元素区域而不是获取整个数组。另外，JNI 提供了不同的函数集以获得数组元素而非其副本的直接指针。

4．对直接指针的操作

可能的话，原生代码可以用 Get <Type>ArrayElements 函数获取指向数组元素的直接指针。如程序清单 3-16 所示，函数带有三个参数，第三个参数是可选参数，该可选参数名是 isCopy，让调用者确定返回的 C 字符串地址指向副本还是指向堆中的固定对象。

程序清单 3-16　获得指向 Java 数组元素的直接指针

```
jint* nativeDirectArray;
jboolean isCopy;

nativeDirectArray = (*env)->GetIntArrayElements(env, javaArray, &isCopy);
```

因为可以像普通的 C 数组一样访问和处理数组元素，因此 JNI 没提供访问和处理数组元素的方法，JNI 要求原生代码用完这些指针立即释放，否则会出现内存溢出。原生代码可以使用 JNI 提供的 Release<Type>ArrayElemens 函数释放 Get<Type>ArrayElements 函数返回的 C 数组，如程序清单 3-17 所示。

程序清单 3-17　释放指向 Java 数组元素的直接指针

```
(*env)->ReleaseIntArrayElements(env, javaArray, nativeDirectArray, 0);
```

该函数带有四个函数，第四个参数是释放模式，表 3-3 列出了支持的释放模式列表。

表 3-3　支持的释放模式

释 放 模 式	动　　作
0	将内容复制回来并释放原生数组
JNI_COMMIT	将内容复制回来但是不释放原生数组，一般用于周期性地更新一个 Java 数组
JNI_ABORT	释放原生数组但不用将内容复制回来

3.4.3　NIO 操作

原生 I/O (NIO)在缓冲管理区、大规模网络和文件 I/O 及字符集支持方面的性能有所改进。JNI 提供了在原生代码中使用 NIO 的函数。与数组操作相比，NIO 缓冲区的数据传送性能较好，更适合在原生代码和 Java 应用程序之间传送大量数据。

1．创建直接字节缓冲区

原生代码可以创建 Java 应用程序使用的直接字节缓冲区，该过程是以提供一个原生 C 字节数组为基础，程序清单 3-18 中列出了 NewDirectByteBuffer 的使用。

程序清单 3-18　基于给定的 C 字节数组创建字节缓冲区

```
unsigned char* buffer = (unsigned char*) malloc(1024);
...
```

```
jobject directBuffer;
directBuffer = (*env)->NewDirectByteBuffer(env, buffer, 1024);
```

注意

原生方法中的内存分配超出了虚拟机的管理范围，且不能用虚拟机的垃圾回收器回收原生方法中的内存。原生函数应该通过释放未使用的内存分配以避免内存泄漏来正确管理内存。

2. 直接字节缓冲区获取

Java 应用程序中也可以创建直接字节缓冲区，在原生代码中调用 GetDirectBufferAddress 函数可以获得原生字节数组的内存地址，如程序清单 3-19 所示。

程序清单 3-19　通过 Java 字节缓冲区获取原生字节数组

```
unsigned char* buffer;
buffer = (unsigned char*) (*env)->GetDirectBufferAddress(env,
    directBuffer);
```

3.4.4　访问域

Java 有两类域：实例域和静态域。类的每个实例都有自己的实例域副本，而一个类的所有实例共享同一个静态域。

JNI 提供了访问两类域的函数，程序清单 3-20 显示了带有静态域和实例域的 Java 类示例。

程序清单 3-20　带有静态域和实例域的 Java 类

```
public class JavaClass {
    /** 实例域 */
    private String instanceField = "Instance Field";

    /** 静态域*/
    private static String staticField = "Static Field";

    ...
}
```

1. 获取域 ID

JNI 提供了用域 ID 访问两类域的方法，可以通过给定实例的 class 对象获取域 ID，用 GetObjectClass 函数可以获得 class 对象，如程序清单 3-21 所示。

程序清单 3-21　用对象引用获得类

```
jclass clazz;
clazz = (*env)->GetObjectClass(env, instance);
```

有两个获得域 ID 的函数分别适用于不同类型域，GetFieldId 函数用于获取实例域，如程序清单 3-22 所示。

程序清单 3-22　获取实例域的域 ID

```
    jfieldID instanceFieldId;
    instanceFieldId = (*env)->GetFieldID(env, clazz, "instanceField",
"Ljava/lang/String;");
```

GetStaticFieldId 用于获取静态域 ID，如程序清单 3-23 所示。这两个函数均返回 jfieldID 类型的域 ID。

程序清单 3-23　获得静态域的域 ID

```
    jfieldID staticFieldId;
    staticFieldId = (*env)->GetStaticFieldID(env, clazz,"staticField",
"Ljava/lang/String;");
```

两个函数的最后一个参数是 Java 中表示域类型的域描述符。在上述示例代码中，"Ljava/lang/String" 表明域类型是 String，本章后面的内容将会再次探讨这个问题。

小贴士

为了提高应用程序的性能，可以缓存域 ID。一般总是缓存使用最频繁的域 ID。

2. 获取域

在获得域 ID 之后，可以用 Get<*Type*>Field 函数获得实际的实例域，如程序清单 3-24 所示。

程序清单 3-24　获得实例域

```
    jstring instanceField;
    instanceField = (*env)->GetObjectField(env, instance, instanceFieldId);
```

用 GetStatic<*Type*>Field 函数获得静态域，如程序清单 3-25 所示。

程序清单 3-25　获得静态域

```
    jstring staticField;
    staticField = (*env)->GetStaticObjectField(env, clazz, staticFieldId);
```

在内存溢出的情况下，这些函数均返回 NULL，此时原生代码不会继续执行。

小贴士

获得单个域值需要调用两到三个 JNI 函数，原生代码回到 Java 中获取每个单独的域值，这给应用程序增加了额外的负担，进而导致性能下降。强烈建议将所有需要的参数传递给原生方法调用，而不是让原生代码回到 Java 中。

3.4.5　调用方法

与域一样，Java 中有两类方法：实例方法和静态方法。JNI 提供访问两类方法的函数，程序清单 3-26 给出了含有一个静态方法和一个实例方法的 Java 类。

程序清单 3-26　带有静态方法和实例方法的 Java 类

```
public class JavaClass {
    /**
     * 实例方法.
     */
    private String instanceMethod() {
        return "Instance Method";
    }

    /**
     * 静态方法.
     */
    private static String staticMethod() {
        return "Static Method";
    }

    ...
}
```

1. 获取方法 ID

JNI 提供了用方法 ID 访问两类方法的途径，可以用给定实例的 class 对象获得方法 ID。用 GetMethodID 函数获得实例方法的方法 ID，如程序清单 3-27 所示。

程序清单 3-27　获得实例方法的方法 ID

```
jmethodID instanceMethodId;
instanceMethodId = (*env)->GetMethodID(env, clazz,
"instanceMethod", "()Ljava/lang/String;");
```

用 GetStaticMethodID 函数获得静态域的方法 ID，如程序清单 3-28 所示。两个函数均返回 jmethodID 类型的方法 ID。

程序清单 3-28　获得静态方法的方法 ID

```
jmethodID staticMethodId;
staticMethodId = (*env)->GetStaticMethodID(env, clazz,
    "staticMethod", "()Ljava/lang/String;");
```

与字段 ID 获取方法一样，两个函数的最后一个参数均表示方法描述符，在 Java 中它表示方法签名。

小贴士

为了提升应用程序的性能，可以缓存方法 ID。一般总是缓存使用最频繁的方法 ID。

2. 调用方法

可以以方法 ID 为参数通过 Call<*Type*>Method 类函数调用实际的实例方法，如程序清单 3-29 所示。

程序清单 3-29　调用实例方法

```
jstring instanceMethodResult;
instanceMethodResult = (*env)->CallStringMethod(env,
    instance, instanceMethodId);
```

用 CallStatic<*Type*>Field 类函数调用静态方法，如程序清单 3-30 所示。

程序清单 3-30　调用静态方法

```
jstring staticMethodResult;
staticMethodResult = (*env)->CallStaticStringMethod(env,
    clazz, staticMethodId);
```

在内存溢出的情况下，这些函数均返回 NULL，此时原生代码不会继续执行。

小贴士

Java 和原生代码之间的转换是代价较大的操作，强烈建议规划 Java 代码和原生代码的任务时考虑这种代价，最小化这种转换可以大大提高应用程序的性能。

3.4.6　域和方法描述符

正如前面几节所提到的，获取域 ID 和方法 ID 均分别需要域描述符和方法描述符，域描述符和方法描述符均可以通过表 3-4 的 Java 类型签名映射获得。

表 3-4　Java 类型签名映射

Java 类 型	签　　名
Boolean	Z
Byte	B
Char	C
Short	S
Int	I
Long	J
Float	F
Double	D
fully-qualified-class	Lfully-qualified-class;
type[]	[type
method type	(arg-type)ret-type

用类型签名映射手工生成域和方法描述符并让它们与 Java 代码同步是一件非常繁琐的任务。

Java 类文件反汇编程序：javap

JDK 提供的命令行方式下的 Java 类文件反汇编程序称为 javap，该工具可以从编译的类文件中解压缩域和方法描述符。

1. 在命令行方式下运行

在命令行方式下，将<Eclipse Workspace>/com.example.hellojni. HelloJni 设置为当前工作目录，该目录是引入 HelloJni 项目的地方。javap 工具在编译的 Java 类文件上操作，它带有两个参数，分别表示编译的类位置及要反汇编的 Java 类名字，格式如下：

```
javap -classpath bin/classes -p -s com.example.hellojni.HelloJni
```

javap 工具将对 com.example.hellojni.HelloJni 类文件进行反汇编并输出图 3-4 所示的域或方法签名。

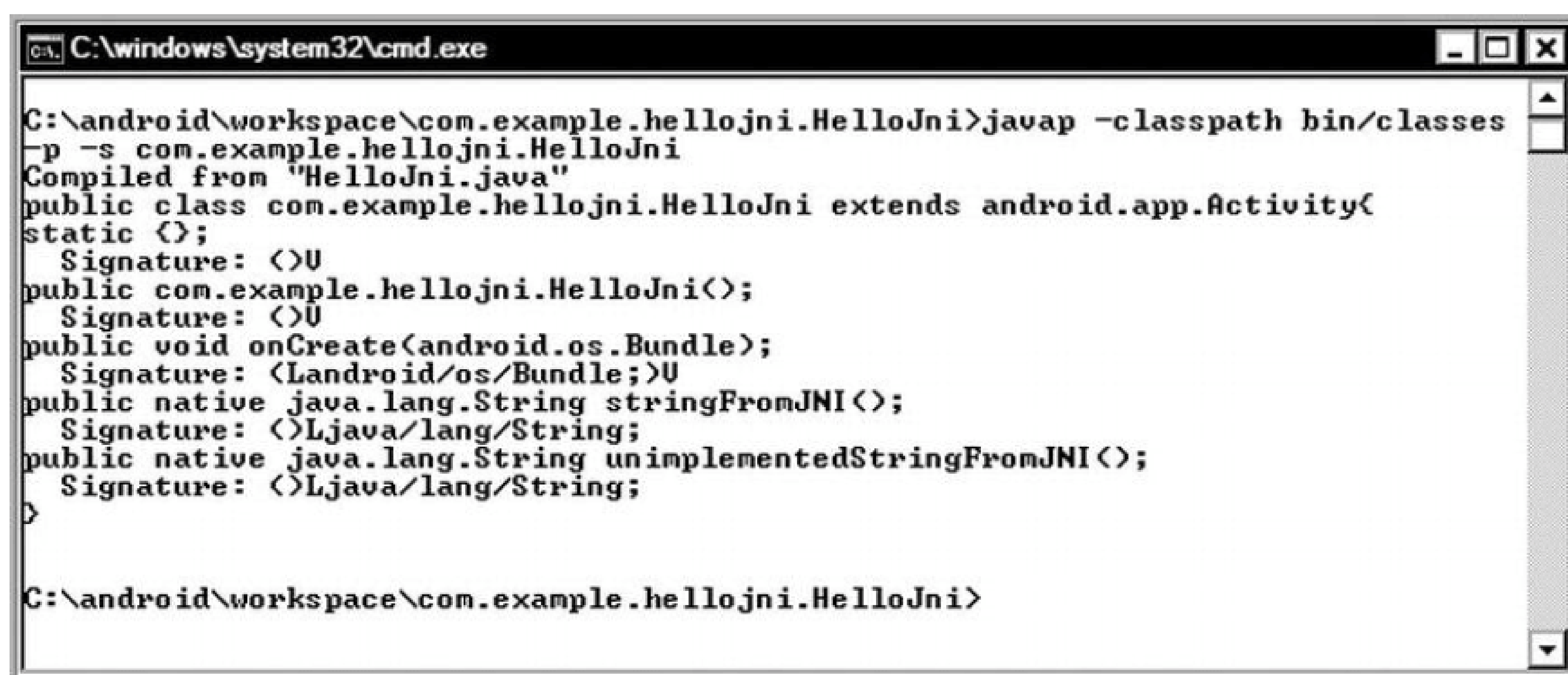


图 3-4　javap 工具输出

与命令行方式每次都运行 javap 工具不同，Eclipse 集成开发环境中将 javap 以外部工具的形式集成在 Eclipse 中以方便用户提取域和方法签名。

2. 在 Eclipse IDE 环境下运行

打开 Eclipse 集成开发环境，在顶部菜单栏中选择 Run | External Tools Configurations…，在 External Tools Configurations 对话框中选择 Program，单击 New launch configurations 按钮。选中 Main 选项卡，如图 3-5 所示，按照如下取值填入工具信息：

- **Name：** Java Class File Disassembler
- **Location：** ${system_path:javap}
- **Working Directory：** ${project_loc}
- **Arguments：** -classpath "${project_classpath};${env_var:ANDROID_SDK_HOME}/platforms/android-14/android.jar" –p –s ${java_type_name}

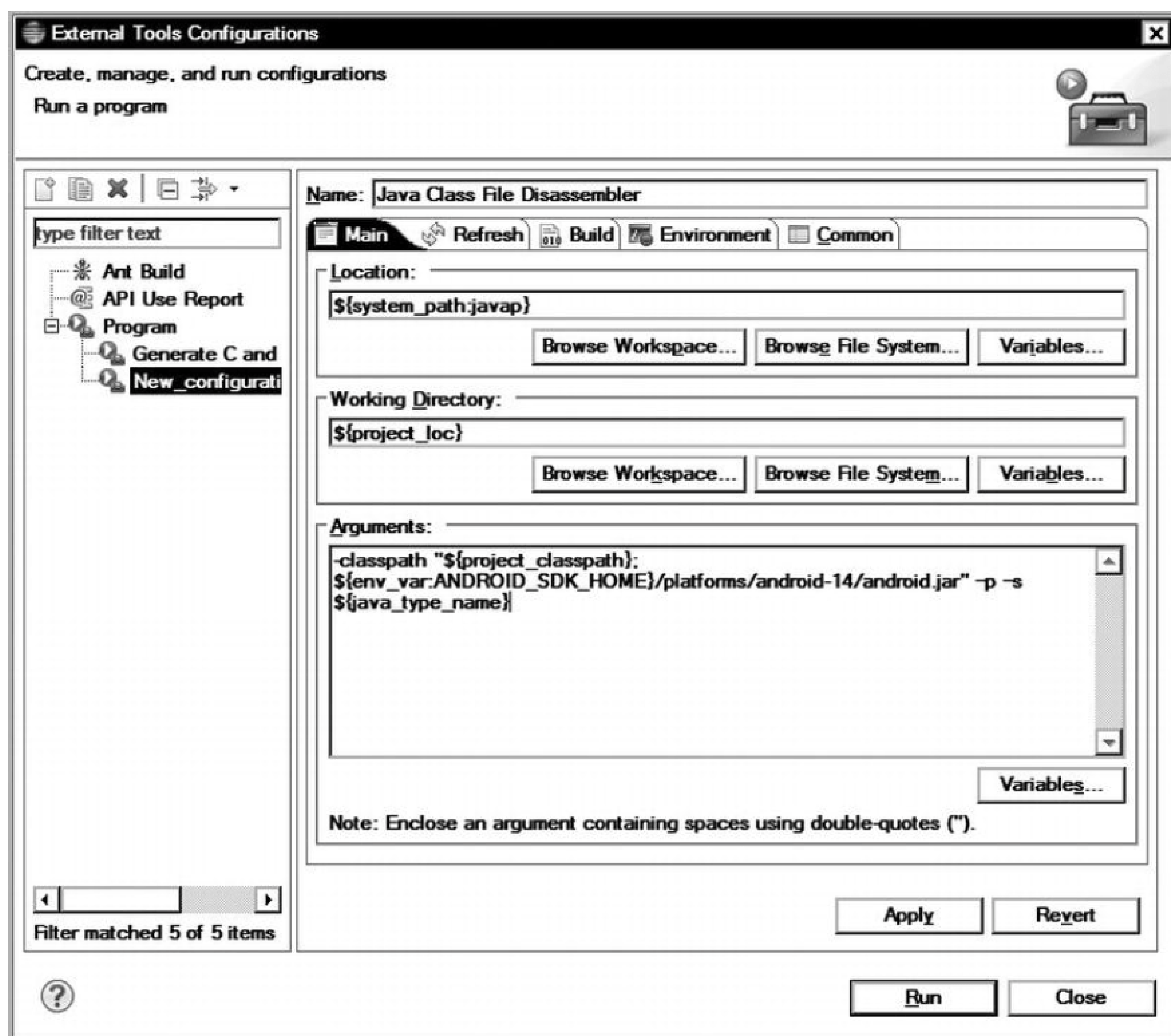


图 3-5　javap 外部工具配置

在 Mac OS X 和 Linux 平台上，用冒号代替分号。切换到 Common 选项卡，像前面介绍过的一样，选中 Display in favorites menu 组下面的复选框 External Tools。

单击 OK 按钮保存外部工具配置。为了测试新配置的效果，在 Project Explorer 视图中选择 HelloJni 类，然后选择 Run | External Tools | Java Class File Disassembler。控制台视图将显示 javah 工具的输出，如图 3-6 所示。

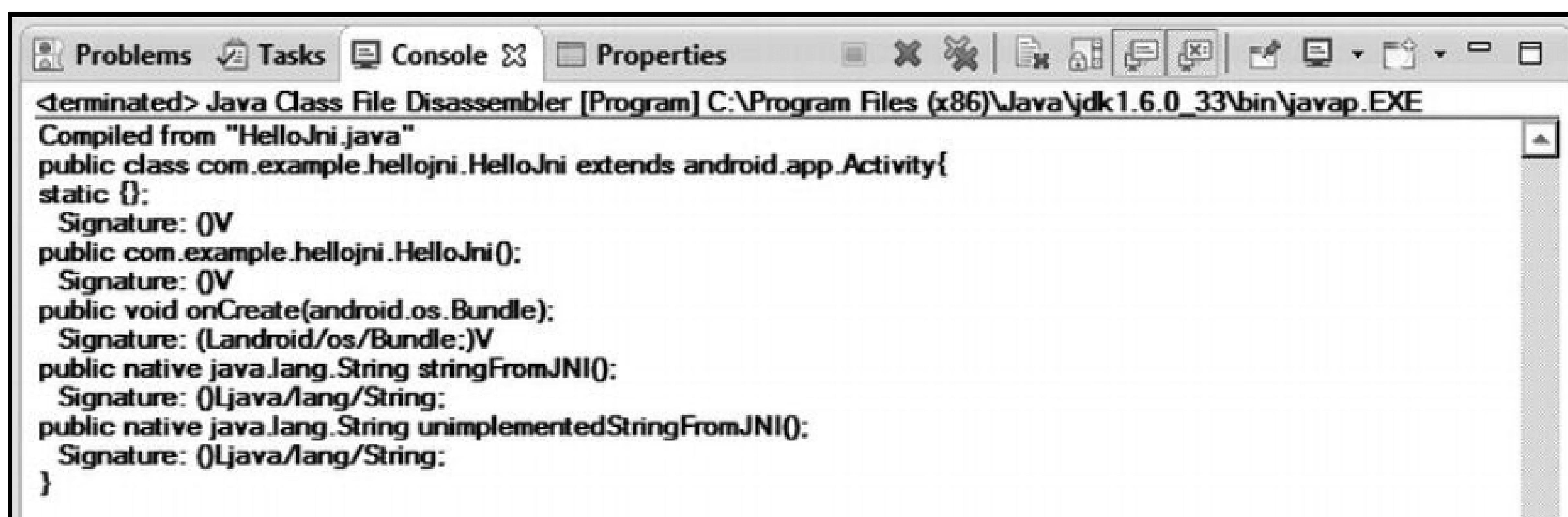


图 3-6　控制台显示 javap 工具的输出

原生代码不易产生异常，处理域和调用 Java 方法可能导致 Java 异常。

3.5　异常处理

异常处理是 Java 程序设计语言的重要功能，JNI 中的异常行为与 Java 中的有所不同，在 Java 中，当抛出一个异常时，虚拟机停止执行代码块并进入调用栈反向检查能处理特定类型异常的异常处理程序代码块，这也叫做捕获异常。虚拟机清除异常并将控制权交给异常处理程序。相比之下，JNI 要求开发人员在异常发生后显式地实现异常处理流。

3.5.1　捕获异常

JNIEnv 接口提供了一组与异常相关的函数集，在运行过程中可以使用 Java 类查看这些函数，以程序清单 3-31 为例。

程序清单 3-31　抛出异常的 Java 例子

```
public class JavaClass {
    /**
     * 抛出方法.
     */
    private void throwingMethod() throws NullPointerException {
        throw new NullPointerException("Null pointer");
    }
    /**
     * 访问方法(原生方法)。
     */
    private native void accessMethods();
}
```

调用 throwingMethod 方法时，accessMethods 原生方法需要显式地做异常处理。JNI 提供了 ExceptionOccurred 函数查询虚拟机中是否有挂起的异常。在使用完之后，异常处理程序需要用 ExceptionClear 函数显式地清除异常，如程序清单 3-32 所示。

程序清单 3-32　原生代码中的异常处理

```
jthrowable ex;
...
(*env)->CallVoidMethod(env, instance, throwingMethodId);
ex = (*env)->ExceptionOccurred(env);
if (0 != ex) {
    (*env)->ExceptionClear(env);

    /* Exception handler. */
}
```

3.5.2　抛出异常

JNI 也允许原生代码抛出异常。因为异常是 Java 类，应该先用 FindClass 函数找到异

常类，用 ThrowNew 函数可以初始化且抛出新的异常，如程序清单 3-33 所示。.

程序清单 3-33　原生代码中抛出异常

```
jclass clazz;
...
clazz = (*env)->FindClass(env, "java/lang/NullPointerException");
if (0 ! = clazz) {
    (*env)->ThrowNew(env, clazz, "Exception message.");
}
```

因为原生函数的代码执行不受虚拟机的控制，因此抛出异常并不会停止原生函数的执行并把控制权转交给异常处理程序。到抛出异常时，原生函数应该释放所有已分配的原生资源，例如内存及合适的返回值等。通过 JNIEnv 接口获得的引用是局部引用且一旦返回原生函数，它们自动地被虚拟机释放。

3.6　局部和全局引用

引用在 Java 程序设计中扮演非常重要的角色。虚拟机通过追踪类实例的引用并收回不再引用的垃圾来管理类实例的使用期限。因为原生代码不是一个管理环境，因此 JNI 提供了一组函数允许原生代码显式地管理对象引用及使用期间原生代码。JNI 支持三种引用：局部引用、全局引用和弱全局引用。下面将详细介绍这几类引用。

3.6.1　局部引用

大多数 JNI 函数返回局部引用。局部引用不能在后续的调用中被缓存及重用，主要因为它们的使用期限仅限于原生方法，一旦原生函数返回，局部引用即被释放。例如：FindClass 函数返回一个局部引用，当原生方法返回时，它被自动释放，也可以用 DeleteLocalRef 函数显式释放原生代码，如程序清单 3-34 所示。

程序清单 3-34　删除一个局部引用

```
jclass clazz;
clazz = (*env)->FindClass(env, "java/lang/String");
...
(*env)->DeleteLocalRef(env, clazz);
```

根据 JNI 的规范，虚拟机应该允许原生代码创建最少 16 个局部引用。在单个方法调用时进行多个内存密集型操作的最佳实践是删除未用的局部引用。如果不可能，原生代码可以在使用之前用 EnsureLocalCapacity 方法请求更多的局部引用槽。

3.6.2　全局引用

全局引用在原生方法的后续调用过程中依然有效，除非它们被原生代码显式释放。

1. 创建全局引用

可以用 NewGlobalRef 函数将局部引用初始化为全局引用，如程序清单 3-35 所示。

程序清单 3-35　用给定的局部引用创建全局引用

```
jclass localClazz;
jclass globalClazz;
...
localClazz = (*env)->FindClass(env, "java/lang/String");
globalClazz = (*env)->NewGlobalRef(env, localClazz);
...
(*env)->DeleteLocalRef(env, localClazz);
```

2. 删除全局引用

当原生代码不再需要一个全局引用时，可以随时用 DeleteGlobalRef 函数释放它，如程序清单 3-36 所示。

程序清单 3-36　删除一个全局引用

```
(*env)->DeleteGlobalRef(env, globalClazz);
```

3.6.3　弱全局引用

全局引用的另一种类型是弱全局引用。与全局引用一样，弱全局引用在原生方法的后续调用过程中依然有效。与全局引用不同，弱全局引用并不阻止潜在的对象被垃圾收回。

1. 创建弱全局引用

可以用 NewWeakGlobalRef 函数对弱全局引用进行初始化，如程序清单 3-37 所示。

程序清单 3-37　用给定的局部引用创建弱全局引用

```
jclass weakGlobalClazz;
weakGlobalClazz = (*env)->NewWeakGlobalRef(env, localClazz);
```

2. 弱全局引用的有效性检验

可以用 IsSameObject 函数检验一个弱全局引用是否仍然指向活动的类实例，如程序清单 3-38 所示.

程序清单 3-38　检验弱全局变量是否仍然有效

```
if (JNI_FALSE == (*env)->IsSameObject(env, weakGlobalClazz, NULL)) {
    /* 对象仍然处于活动状态且可以使用*/
} else {
    /* 对象被垃圾回收器收回，不能使用*/
}
```

3. 删除弱全局引用

可以随时用 DeleteWeakGlobalRef 函数释放弱全局引用，如程序清单 3-39 所示。

程序清单 3-39　删除一个弱全局引用

```
(*env)->DeleteWeakGlobalRef(env, weakGlobalClazz);
```

全局引用显式释放前一直有效，它们可以被其他原生函数及原生线程使用。

3.7　线程

作为多线程环境的一部分，虚拟机支持运行的原生代码。在开发原生构件时要记住 JNI 技术的一些约束：

- 只在原生方法执行期间及正在执行原生方法的线程环境下局部引用是有效的，局部引用不能在多线程间共享，只有全局引用可以被多个线程共享。
- 被传递给每个原生方法的 JNIEnv 接口指针在与方法调用相关的线程中也是有效的，它不能被其他线程缓存或使用。

3.7.1　同步

同步是多线程程序设计最重要的特征。与 Java 的同步类似，JNI 的监视器允许原生代码利用 Java 对象同步，虚拟机保证存取监视器的线程能够安全执行，而其他线程等待监视器对象变成可用状态。Java 应用程序中的同步如程序清单 3-40 所示。

程序清单 3-40　Java 同步代码块

```
synchronized(obj) {
    /*同步线程安全代码块。 */
}
```

在原生代码中，相同级别同步可以用 JNI 的监视器方法实现的，如程序清单 3-41 所示。

程序清单 3-41　Java 同步代码块的原生等价

```
if (JNI_OK == (*env)->MonitorEnter(env, obj)) {
    /* 错误处理 */
}

/* 同步线程安全代码块。*/

if (JNI_OK == (*env)->MonitorExit(env, obj)) {
    /* 错误处理. */
}
```

注意

对 MonitorEnter 函数的调用应该与对 MonitorExit 的调用相匹配，从而避免代码出现死锁。

3.7.2　原生线程

为了执行特定任务，这些原生构件可以并行使用原生线程。因为虚拟机不知道原生线程，因此它们不能与 Java 构件直接通信。为了与应用的依然活跃部分交互，原生线程应该先附着在虚拟机上。

JNI 通过 JavaVM 接口指针提供了 AttachCurrentThread 函数以便于让原生代码将原生线程附着到虚拟机上，如程序清单 3-42 所示，JavaVM 接口指针应该尽早被缓存，否则的话它不能被获取。

程序清单 3-42　将当前线程与虚拟机附着和分离

```
JavaVM* cachedJvm;
...
JNIEnv* env;
...
/* 将当前线程附着到虚拟机. */
(*cachedJvm)->AttachCurrentThread(cachedJvm, &env, NULL);

/* 可以用 JNIEnv 接口实现线程与 Java 应用程序的通信*/

/* 将当前线程与虚拟机分离 */
(*cachedJvm)->DetachCurrentThread(cachedJvm);
```

对 AttachCurrentThread 函数的调用允许应用程序获得对当前线程有效的 JNIEnv 接口指针。将一个已经附着的原生线程再次附着不会有任何副作用。当原生线程完成时，可以用 DetachCurrentThread 函数将原生线程与虚拟机分离。

3.8　小结

本章介绍了用 JNI 技术实现 Java 应用程序与原生代码之间通信的方法，关于 JNI 技术的更多信息和可用的 JNI API 可以访问 http://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/jniTOC.html 网站，在 Oracle 的 JNI 文档找查到。

正如你已经看到的，对 JNI 所作的任何操作需要调用 2～3 个函数，实现大量的原生方法并让他们与 Java 类同步很容易成为一个繁杂的任务。第 4 章将学习能够基于已经存在的代码接口自动生成 JNI 代码的开放源代码的解决方案。

第 4 章

使用 SWIG 自动生成 JNI 代码

第 3 章介绍了 JNI 技术以及将原生代码和 Java 应用程序连接的方法。如前所述，实现 JNI 封装代码和处理数据类型之间的转换是烦琐且耗时的开发任务。本章将介绍简化的包装器和接口生成器(SWIG，Simplified Wrapper and Interface Generator)，SWIG 是可以通过自动生成必要的 JNI 封装代码来简化上述过程的开发工具。

SWIG 不是 Android 或 Java 的专用工具。它是一个可以生成许多其他编程语言代码的、广泛使用的工具。由于 SWIG 十分庞大，本章只会介绍下列能让你初步了解 SWIG 的主要概念和 API：

- 为原生代码定义 SWIG 接口
- 基于所定义的接口生成 JNI 代码
- 将 SWIG 集成到 Android 构建过程中
- 包装 C/C++代码
- 异常处理
- 使用内存管理
- 在原生代码中调用 Java 程序

由于 SWIG 简化了 JNI 代码的开发，在以后几章中会经常用到 SWIG。

4.1 什么是 SWIG

SWIG 是一个编译时软件开发工具，它能生成将用 C/C++编写的原生模块与包括 Java 在内的其他编程语言进行联接的必要代码。SWIG 不仅是一个代码生成器，还是一个接口编译器。它不定义新的协议，也不是一个组件框架或者一个特定的运行时库。SWIG 把接口文件看做输入，并生成必要的代码在 Java 中展示接口，从而让 Java 能够理解原生代码中的接口定义。SWIG 不是一个存根生成器；它产生将要被编译和运行的代码。

SWIG 最初是在 1995 年用于科学计算的应用而开发出来的，现在它已经发展成在 GNU GPL 开源许可下发布的通用工具。想了解 SWIG 的更多相关信息，请登录 www.swig.org。

4.2 安装

SWIG 可应用于包括 Windows、Mac OS X 和 Linux 在内的大多数操作系统平台。本书撰写时，SWIG 的最新版本是 2.0.7，其代码在其官方网站 www.swig.org 上以源代码包的形式提供。除了 Windows 二进制文件以外，其他操作系统上的 SWIG 二进制文件都通过特定操作系统库提供。本节将详细介绍几个主要操作系统中 SWIG 二进制文件的下载方式和安装指令。

4.2.1 Windows 平台上 SWIG 的安装

Windows 平台上 SWIG 二进制文件通过 www.swig.org/download.html 网站的 SWIG 下载页面下载。如图 4-1 所示，单击链接下载 SWIG 安装包。

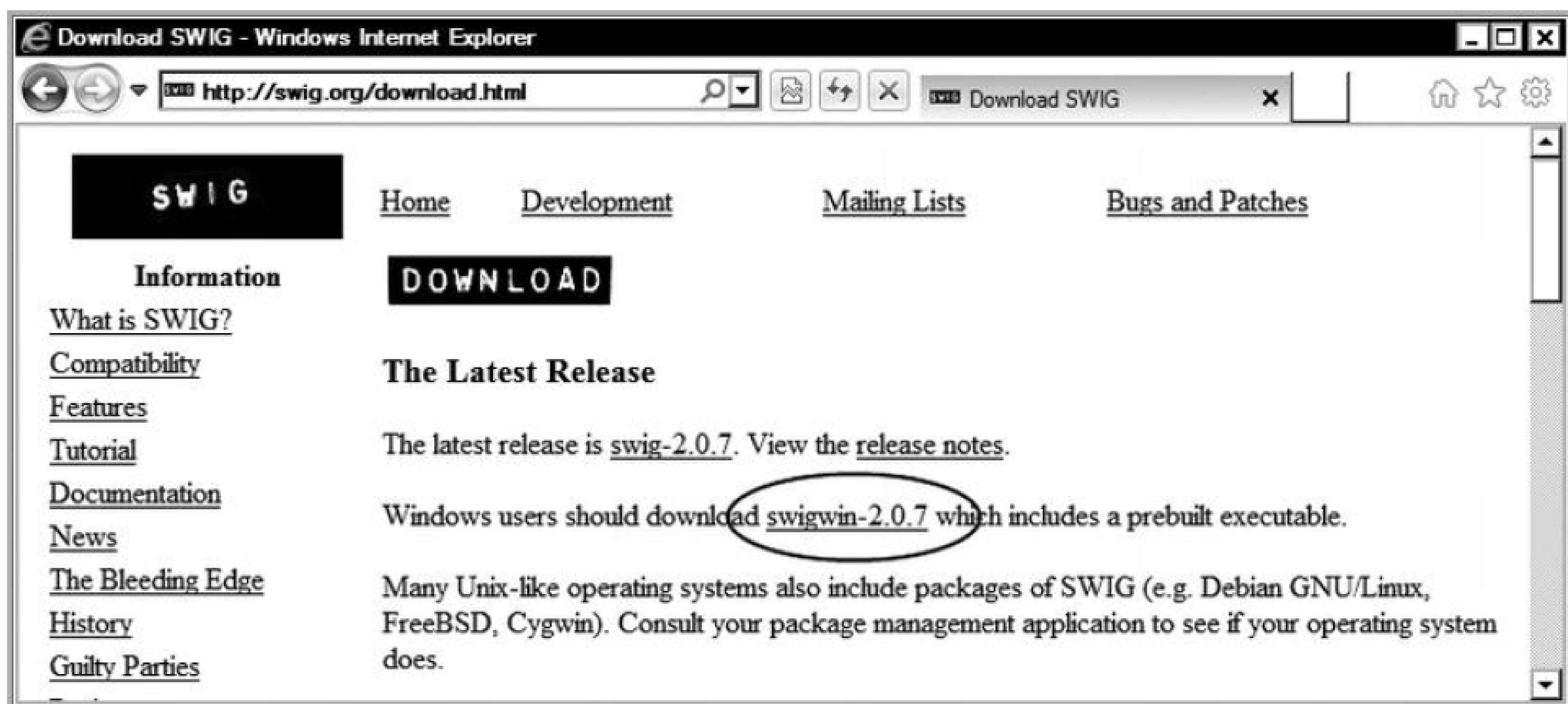


图 4-1 Windows 下 SWIG 下载链接

SWIG 安装包是一个 ZIP 格式的文件，Windows 操作系统支持 ZIP 格式的文件。下载完成后，右击 ZIP 文件，并在上下文菜单中选择 Extract All 打开 Extract Compressed Folder 向导对话框。单击 Browse 按钮，选择 SWIG 文件的目标目录。第 1 章已经讲过，C:\android 目录是用于保存开发工具的根目录，因此将 C:\android 选作目标目录。因为 ZIP 文件中已经包含一个名字为 swigwin-2.0.7 的子目录，其中包含 Android 的 SWIG 文件，因此不需要建立专用的空白目标目录。单击 Extract 按钮开始安装。

和已经安装的其他开发工具类似，为了方便访问 SWIG，应该将其安装目录添加到系统的可执行文件搜索路径中。从 System Properties 中打开 Environment Variables 对话框，单

击 New 按钮。如图 4-2 所示，在 New System Variable 对话框中，将变量名设置为 SWIG_HOME，变量值设置为 SWIG 的安装目录，如 C:\android\swigwin-2.0.7。

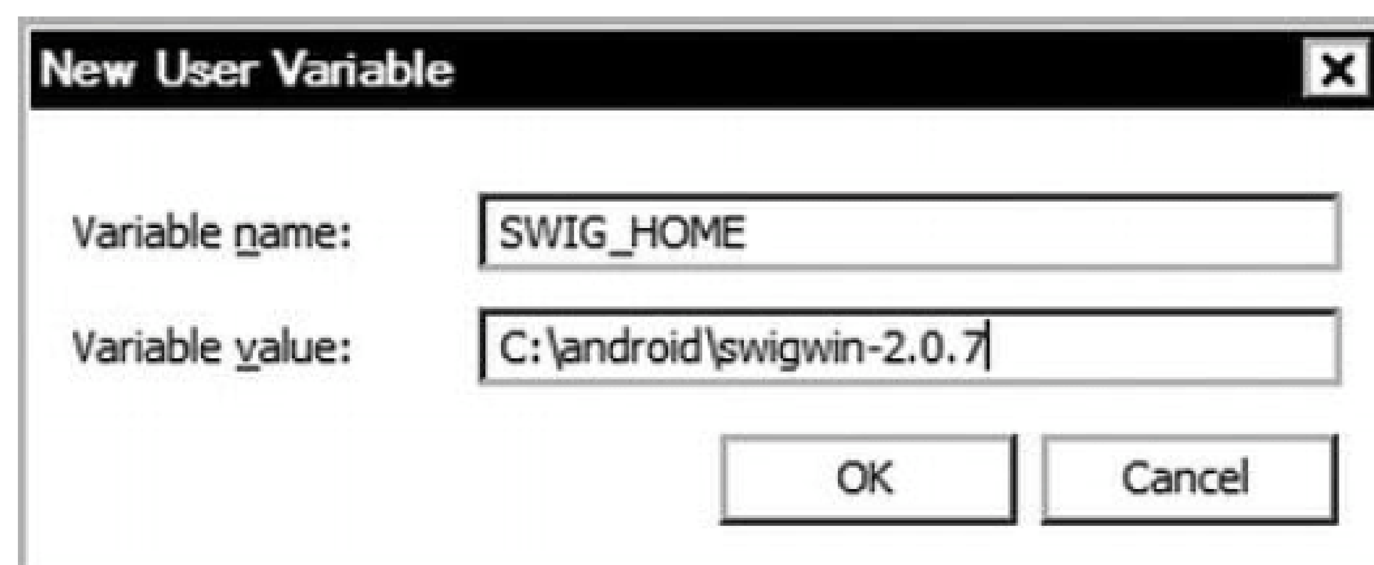


图 4-2　新建 SWIG_HOME 环境变量

在系统变量列表中双击 PATH 变量，在变量值后添加;%SWIG_HOME%，如图 4-3 所示。

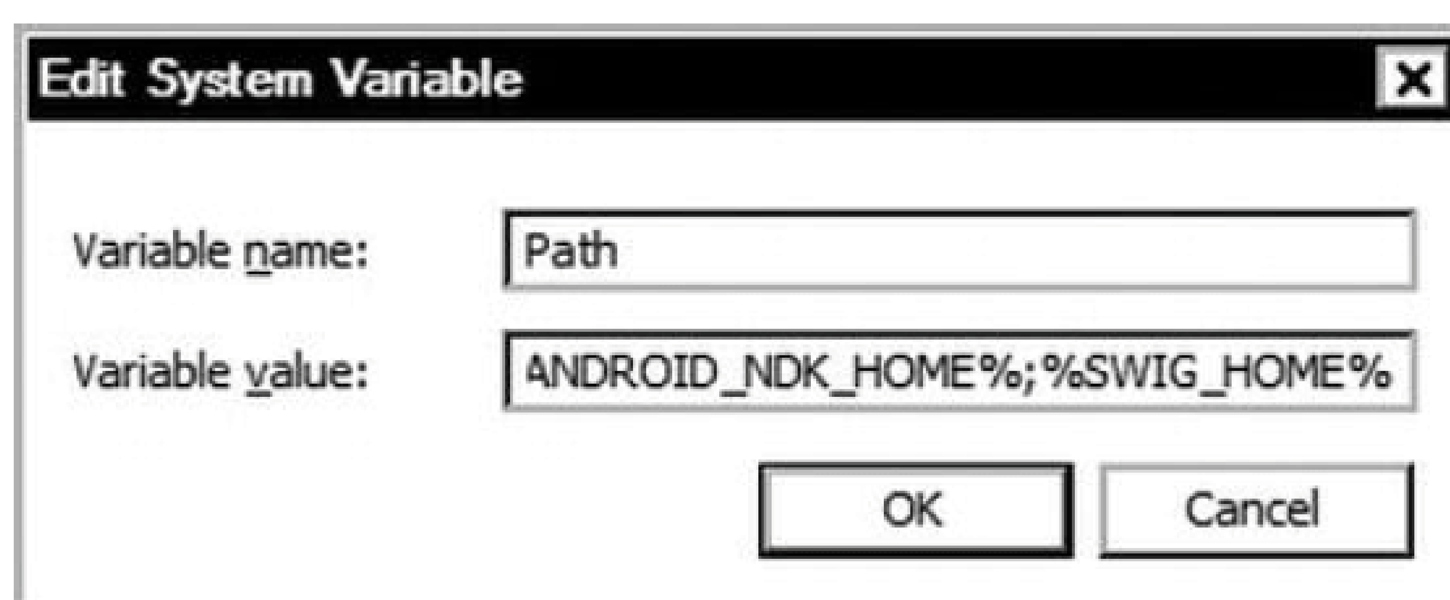


图 4-3　向系统 PATH 变量中追加 SWIG 二进制文件路径

如果安装成功，你将看到 SWIG 版本号，如图 4-4 所示。

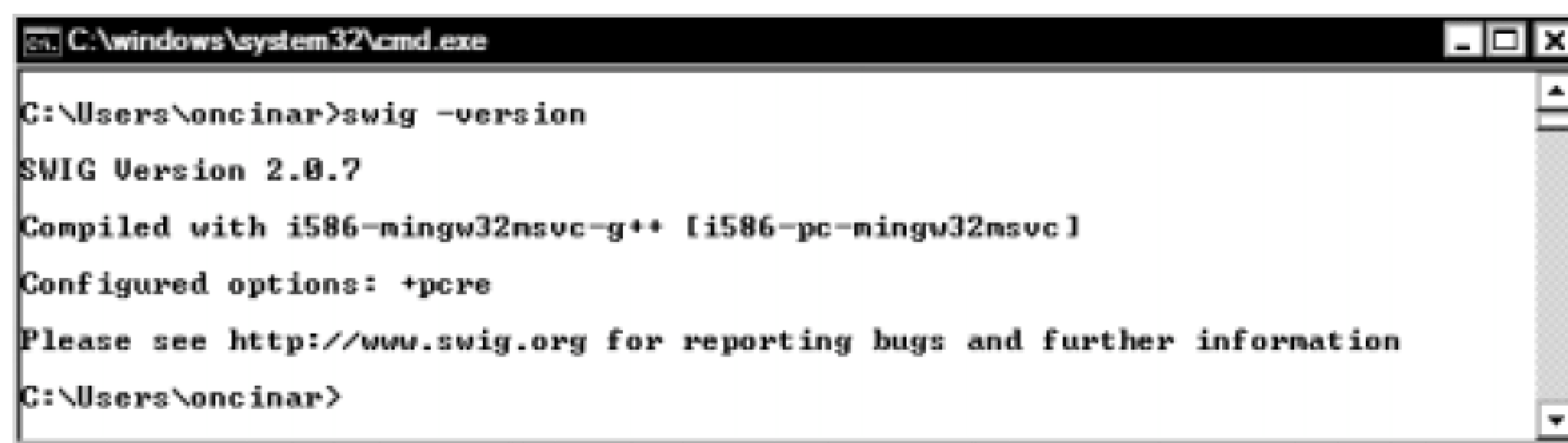


图 4-4　验证 SWIG 安装效果

4.2.2　在 Mac OS X 下安装

SWIG 网站上没有提供 Mac OS X 平台安装包，用 Homebrew 包管理器下载并安装 SWIG。为了使用 Homebrew，需要将其装在主机上。Homebrew 是一个基于控制台的安装应用程序，从 Homebrew 安装页面上复制安装指令，如图 4-5 所示。

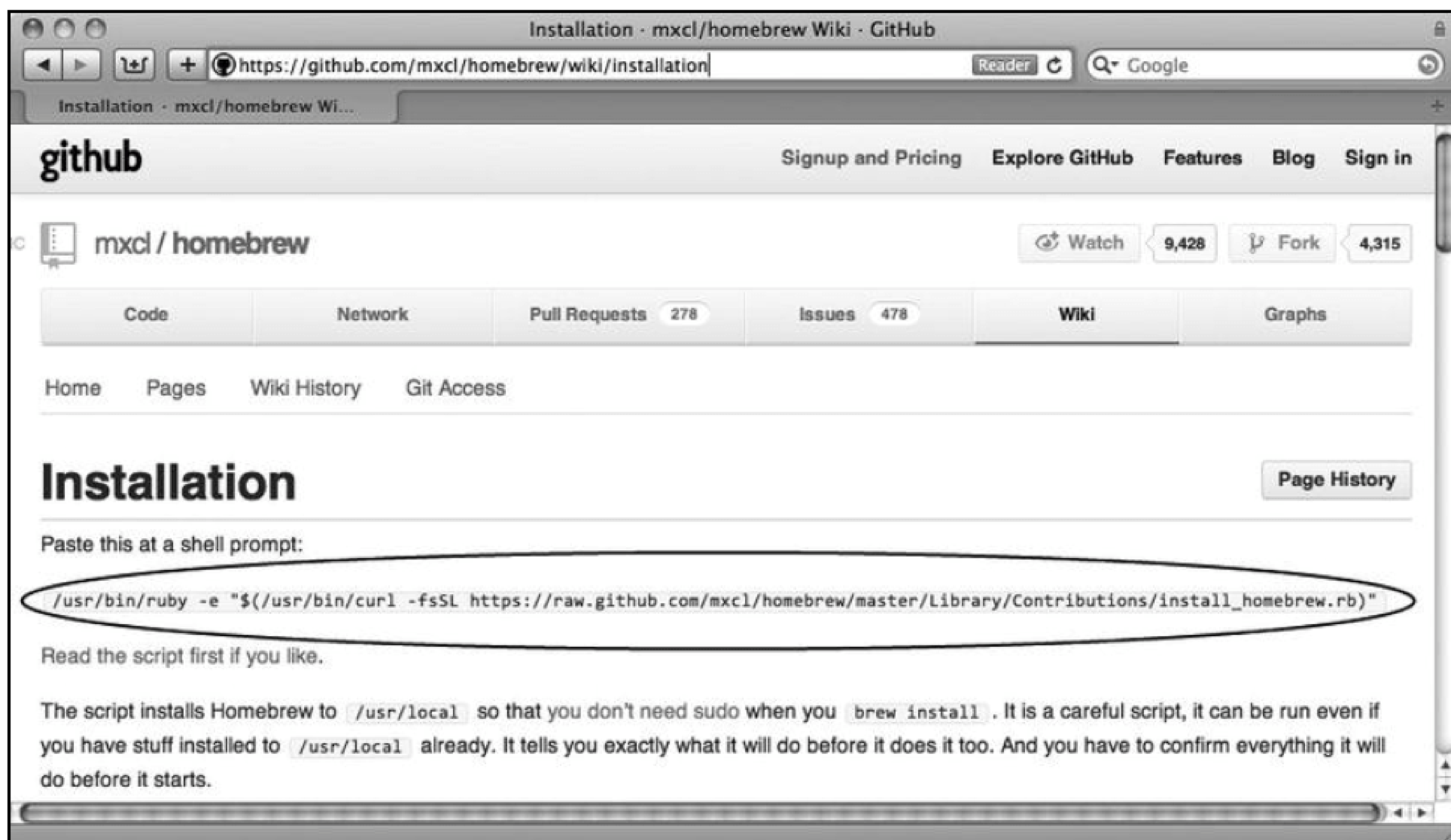


图 4-5 Homebrew 安装命令

打开一个终端窗口，在命令提示符下粘贴安装命令，如图 4-6 所示，然后按回车键开始安装。该命令首先下载 Homebrew 安装脚本 ，然后用 Ruby 执行该脚本，按照屏幕上的指令完成安装过程。

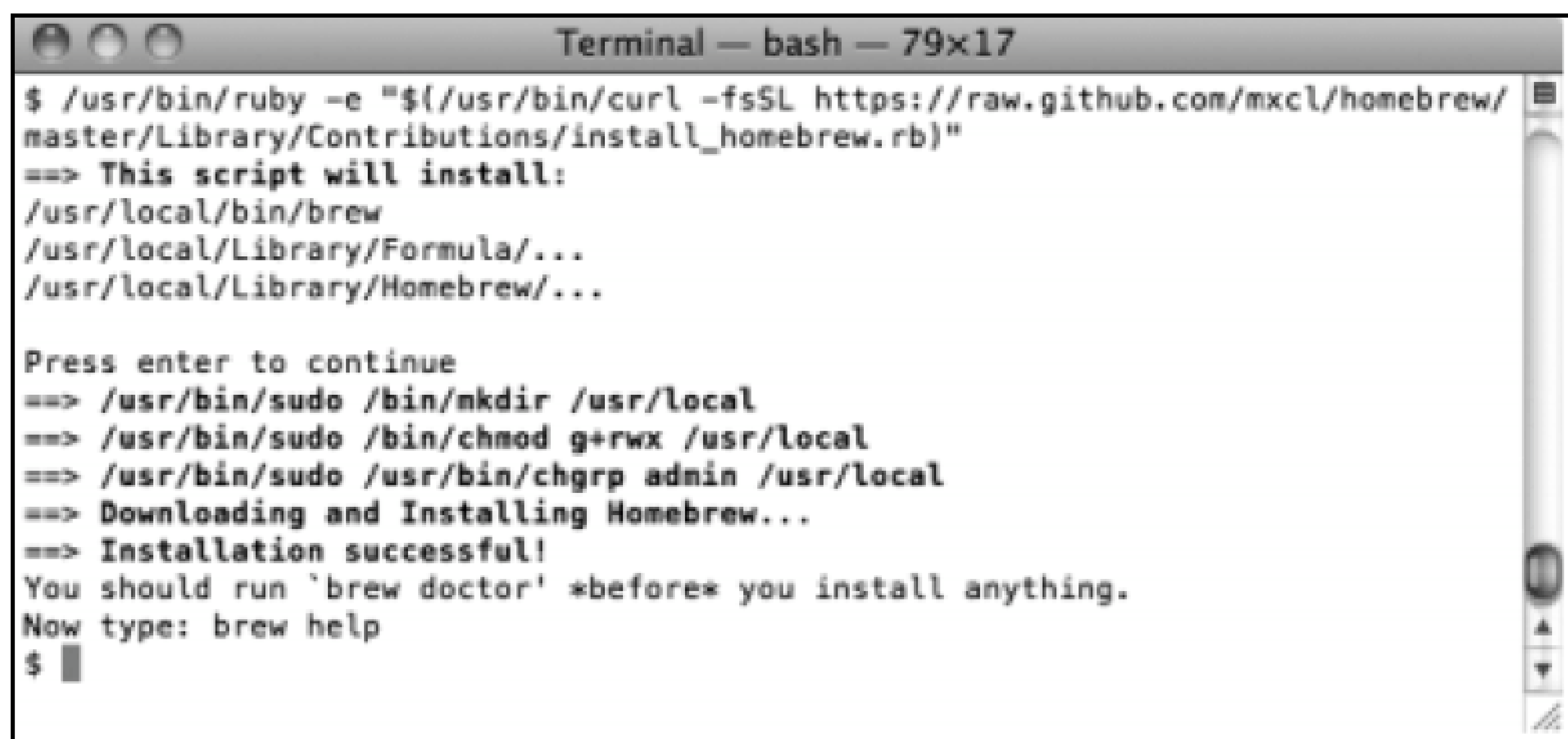


图 4-6 从命令行安装 Homebrew

安装完 Homebrew 就可以安装 SWIG 了。打开终端窗口，在命令提示符下执行 brew install swig，如图 4-7 所示。Homebrew 将下载 SWIG 的源代码及其扩展工具，然后自动编译并安装 SWIG。

为了验证安装是否成功，打开一个新的终端窗口，在命令行方式下执行 swig -version。如果安装成功，会看到 SWIG 版本号，如图 4-8 所示。

```
$ brew install swig
==> Installing swig dependency: pcre
==> Downloading ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/pcre-8.31
############################################################################ 100.0%
==> ./configure --prefix=/usr/local/Cellar/pcre/8.31 --enable-utf8 --enable-uni
==> make test
==> make install
/usr/local/Cellar/pcre/8.31: 130 files, 3.2M, built in 54 seconds
==> Installing swig
==> Downloading http://downloads.sourceforge.net/project/swig/swig/swig-2.0.7/s
############################################################################ 100.0%
==> ./configure --prefix=/usr/local/Cellar/swig/2.0.7
==> make
==> make install
/usr/local/Cellar/swig/2.0.7: 598 files, 6.2M, built in 66 seconds
$
```

图 4-7　使用 Homebrew 安装 SWIG

```
$ swig -version

SWIG Version 2.0.7

Compiled with /usr/bin/g++-4.2 [i386-apple-darwin10.8.0]

Configured options: +pcre

Please see http://www.swig.org for reporting bugs and further information
$
```

图 4-8　验证 SWIG 是否安装成功

4.2.3　在 Ubuntu Linux 下安装

SWIG 网站没提供 Linux 平台的安装包，Ubuntu Linux 软件资源库包含最新版本的 SWIG，可以用系统包管理器安装 SWIG。再打开一个终端窗口，在命令提示符下执行 sudo apt-get install swig，如图 4-9 所示。系统包管理器将自动下载并安装 SWIG 及其扩展工具。

```
$ sudo apt-get install swig
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  swig2.0
Suggested packages:
  swig-doc swig-examples swig2.0-examples swig2.0-doc
The following NEW packages will be installed:
  swig swig2.0
0 upgraded, 2 newly installed, 0 to remove and 9 not upgraded.
Need to get 0 B/1,120 kB of archives.
After this operation, 4,370 kB of additional disk space will be used.
Do you want to continue [Y/n]? Y
Selecting previously unselected package swig2.0.
(Reading database ... 160649 files and directories currently installed.)
Unpacking swig2.0 (from .../swig2.0_2.0.4+really2.0.4-4ubuntu2_i386.deb) ...
Selecting previously unselected package swig.
Unpacking swig (from .../swig_2.0.4+really2.0.4-4ubuntu2_i386.deb) ...
Processing triggers for man-db ...
Setting up swig2.0 (2.0.4+really2.0.4-4ubuntu2) ...
Setting up swig (2.0.4+really2.0.4-4ubuntu2) ...
$
```

图 4-9　在命令行方式下安装 SWIG

为了验证安装是否成功，打开一个新的终端窗口，在命令行行方式下执行 swig –version。如果安装成功，你将看到 SWIG 版本号，如图 4-10 所示。

```
$ swig -version

SWIG Version 2.0.4

Compiled with g++ [i686-pc-linux-gnu]

Configured options: +pcre

Please see http://www.swig.org for reporting bugs and further information
$
```

图 4-10　验证 SWIG 是否安装成功

4.3　通过示例程序试用 SWIG

在深入学习 SWIG 之前，通过一个示例程序我们可以更好地理解 SWIG 的工作方法。Android 平台是建立在 Linux 操作系统之上的多用户平台，它在虚拟机沙箱上运行应用程序并把它们看成系统上的不同用户以保证平台安全。在 Linux 系统中，给每个用户分配一个用户 ID，可以用 POSIX OS API 的 getuid 函数查询这个用户 ID。作为一个平台独立的编程语言，Java 不提供对这些函数的访问。作为该示例应用程序的一部分，我们要做以下几件事：

- 写一个 SWIG 接口文件以展示 getuid 函数
- 将 SWIG 集成到 Android 构建过程中
- 将 SWIG 生成的源文件加入 Android.mk 构建文件中
- 用 SWIG 生成的代理类查询 getuid
- 在屏幕上显示结果

你将用 hello-jni 这个示例项目进行测试。打开 Eclipse IDE，进入 hello-jni 项目。如前所述，在接口文件上操作 SWIG。

4.3.1　接口文件

SWIG 接口文件包含函数原型、类和变量声明，它的语法和普通的 C / C++头文件一样。除了 C / C + +关键字和预处理器指令，接口文件还包含 SWIG 特有的预处理器指令，该指令可用于优化生成封装代码。

为了展示 getuid，需要定义一个接口文件。在 Project Explorer 视图中，右击 hello-jni 项目下的 jni 目录，选择 New | File 打开 New File 对话框。如图 4-11 所示，将文件名设置为 Unix.i，然后单击 Finish 按钮。

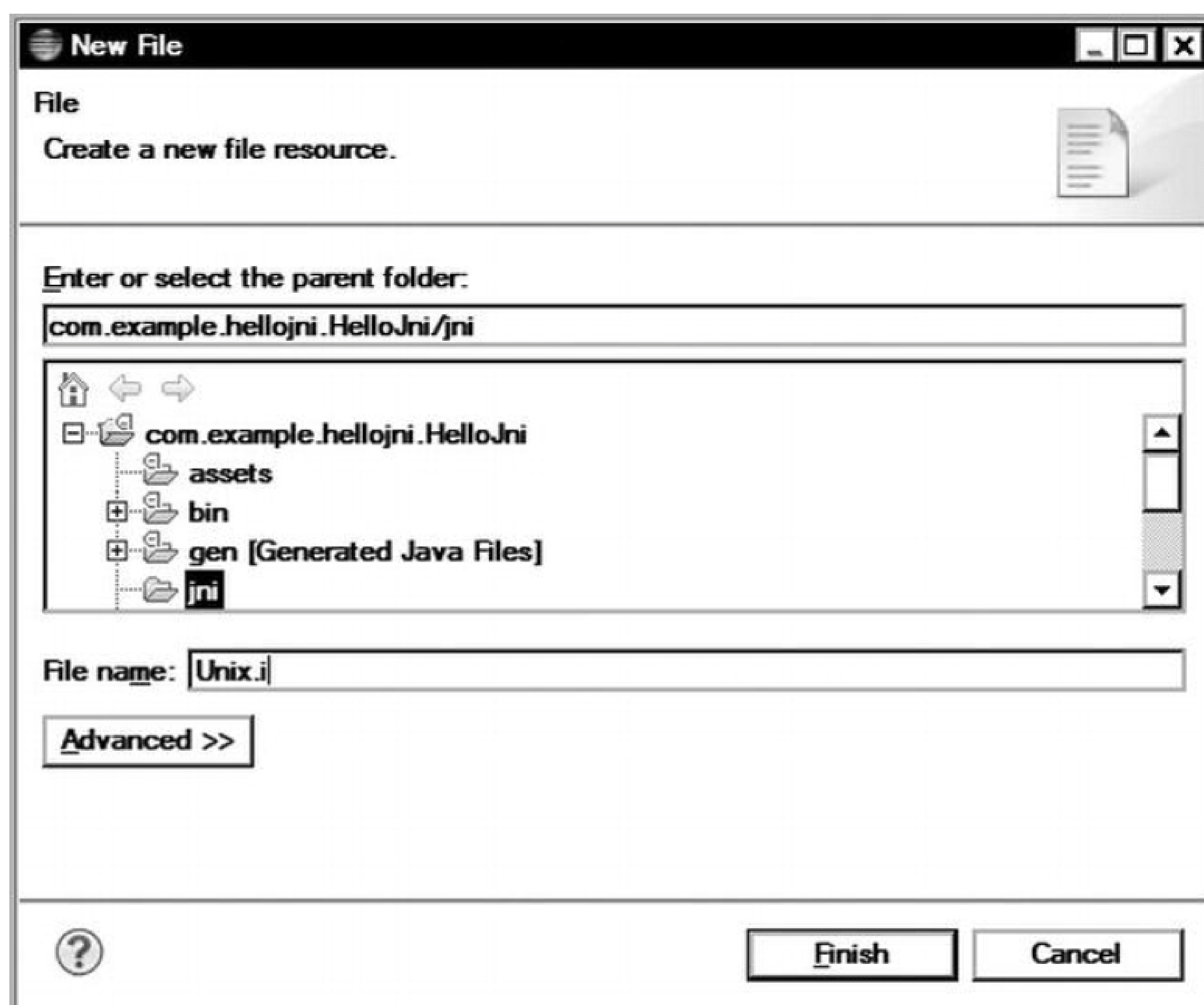


图 4-11　创建 Unix.i 的 SWIG 接口文件

在 Editor 视图中填入 Unix.i 内容，如程序清单 4-1 所示。

程序清单 4-1　Unix.i 接口文件内容

```
/* 模块名是 Unix. */
%module Unix

%{
/* 包含 POSIX 操作系统 API. */
#include < unistd.h>
%}

/* 告诉 SWIG uid_t. */
typedef unsigned int uid_t;

/* 让 SWIG 包装 getuid 函数. */
extern uid_t getuid(void);
```

在进行下一步操作之前(调用 SWIG)，简单介绍一下接口文件。

1. 注释

Unix 接口文件的注释行与 C 的注释行风格相同，以/*开头，以*/结尾，如程序清单 4-12 所示。与编译器一样，SWIG 也不处理注释行，它们只是为开发人员提供对接口文件的注

释说明。

程序清单 4-2　Unix.i 用注释开头

```
/*模块名是 Unix. */
. . .
```

2. 模块名

每次调用 SWIG 都需要指定一个模块名，模块名用于给生成的封装文件命名，用 SWIG 特定的预处理指令%module 指定模块名，该指令放在每个接口文件的开头。遵循这个规则，Unix.i 接口文件也由模块名定义语句开头，比如将模块名定义为 Unix，如程序清单 4-3 所示。

程序清单 4-3　定义 Unix.i 的模块名

```
%module Unix
```

3. 用户定义的代码

SWIG 只在生成封装代码时使用接口文件，文件内容也一样，因此要把用户自定义的代码包含在生成的文件中，比如编译生成的代码所需要的头文件等。当 SWIG 生成封装代码时，代码被分为五个部分，SWIG 提供预处理指令让开发人员指定代码片段哪个部分，SWIG 的预处理指令语法如程序清单 4-4 所示。

程序清单 4-4　SWIG 插入预处理指令的语法

```
% < section > %{
    . . .
   这个代码块将按照 section 的指定包含在生成代码的相应部分
   . . .
%}
. . .
```

< section > 部分内容可以如下：

- **begin**：将代码块放在生成的封装文件的开头位置，主要用于定义文件后面部分使用的预处理器宏。
- **runtime**：将代码块放在 SWIG 的内部类型检查及其他支持函数之后。
- **header**：将代码块放在 header 部分，这部分在头文件和其他帮助函数之后，这是生成的文件中插入代码的默认位置，可以缩写为%{ . . . %}。
- **wrapper**：将代码块放在生成的封装函数之后。
- **init**：将代码块放入装入时初始化模块的函数中。

如程序清单 4-5 所示，Unix.i 接口文件用插入头预处理指令的简短形式在生成的封装代码插入一个头文件。

程序清单 4-5　Unix.i 将一个头文件插入到生成的封装代码

```
%{
/*包含 POSIX 操作系统 API. */
#include < unistd.h>
%}
```

4. 类型定义

SWIG 能理解 C/C++的所有数据类型，但是把其他的东西都看成对象并把它们封装成指针。从 getuid 函数的声明可以看出该函数的返回值类型为 uid_t，它并不是一个标准的 C/C++的数据类型。因此，SWIG 把它看做一个对象并把它封装为一个指针。由于 uid_t 仅仅是一个简单的基于无符号整型的类型名而不是一个对象，因此将它看做对象不合适。如程序清单 4-6 所示，Unix.i 接口文件用一个类型定义让 SWIG 知道 getuid 函数的实际返回值类型。

程序清单 4-6　id_t 的类型定义

```
/* 告诉 SWIG uid_t. */
typedef unsigned int uid_t;
```

5. 函数原型

Unix.i 接口文件以 getuid 函数的函数原型结尾，如程序清单 4-7 所示。

程序清单 4-7　getuid 函数原型

```
/*请求 SWIG 封装 getuid 函数 */
extern uid_t getuid(void);
```

这里简单解释了 SWIG 如何获取生成封装代码的指令，从而在 Java 中展示原生函数。

4.3.2　在命令行方式下调用 SWIG

既然接口准备好了，可以调用 SWIG 以生成必要封装代码，进而在 Java 中展示 getuid 函数。SWIG 将生成两组文件：一是封装 C/C++代码以展示原生函数，二是提供访问被展示函数的代理类。

1. 代理类的 Java 包

应该在调用 SWIG 之前创建 Java 包目录。在 Project Explorer 中右击 src 目录，选择 New | Package 打开 New Java Package 对话框。如图 4-12 所示，设置包名为 com.apress.swig，并单击 Finish 按钮。

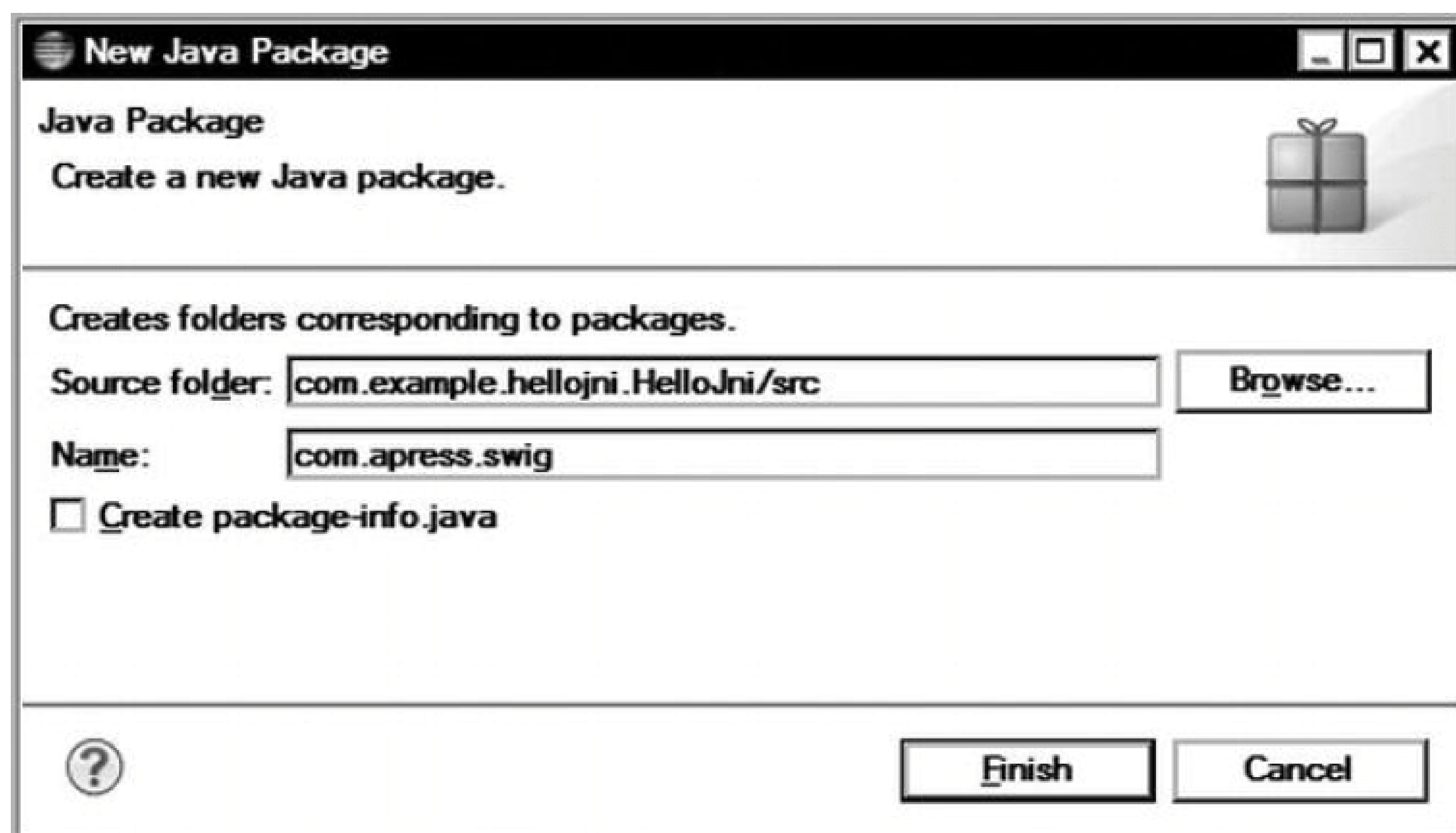


图 4-12　SWIG 文件的 Java 包

2. 调用 SWIG

现在可以调用 SWIG 了。打开一个终端窗口或一个命令提示符，进入 hello-jni 项目的导入目录，例如 C:\android\workspace\com.example.hellojni.HelloJni。如图 4-13 所示，在命令行方式下执行 swig -java -package com.apress.swig -outdir src/com/apress/swig jni/Unix.i 命令。

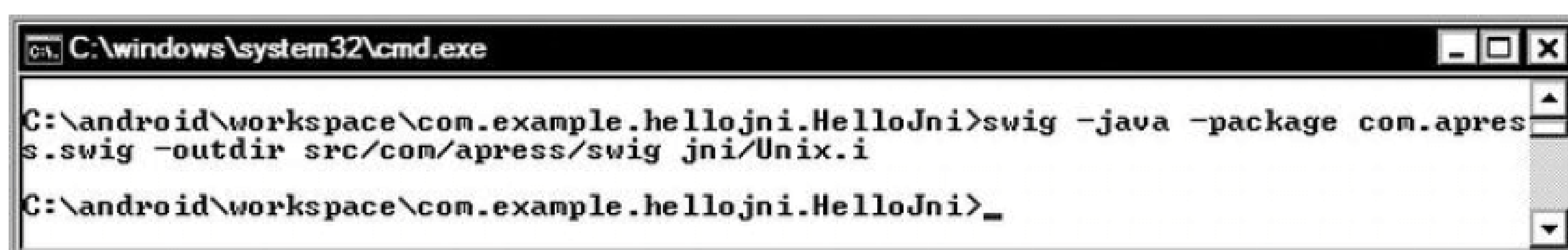


图 4-13　在命令行方式下调用 SWIG

SWIG 解析 Unix.i 接口文件并且在 jni 目录下生成 C/C++ 封装代码 Unix_wrap.c，同时在 com.apress.swig 包中生成 Java 代理类 UnixJNI.java 和 Unix.java。在开始深入研究这些文件之前，先简化一下这个过程。SWIG 可以被集成到 Android 构建过程中，而不需要在命令行方式下手动执行。

4.3.3　将 SWIG 集成到 Android 构建过程中

1. SWIG 的 Android 构建系统

打开 Project Explorer，右击 jni 目录，并在菜单中选择 New | File。打开 New File 对话框，创建一个名为 my-swig-generate.mk 的文件，该 Makefile 片段的内容如程序清单 4-8 所示。

程序清单 4-8　my-swig-generate.mk 文件的内容

```
#
# Android 构建系统的 SWIG 扩展.
```

```
#
# @author Onur Cinar
#

# 检查变量 MY_SWIG_PACKAGE 是否已经定义
ifndef MY_SWIG_PACKAGE
  $(error MY_SWIG_PACKAGE is not defined.)
endif

# 用斜线替换 Java 目录中的圆点
MY_SWIG_OUTDIR:= $(NDK_PROJECT_PATH)/src/$(subst .,/,$(MY_SWIG_PACKAGE))
# SWIG 的默认类型是 C
ifndef MY_SWIG_TYPE
  MY_SWIG_TYPE := c
endif

#设置 SWIG 的模式
ifeq ($(MY_SWIG_TYPE),cxx)
  MY_SWIG_MODE := — c++
else
  MY_SWIG_MODE :=
endif

# 追加 SWIG 封装源文件
LOCAL_SRC_FILES+= $(foreach MY_SWIG_INTERFACE,\
  $(MY_SWIG_INTERFACES),\
  $(basename $(MY_SWIG_INTERFACE))_wrap.$(MY_SWIG_TYPE))

# 添加.cxx 作为 C++扩展名
LOCAL_CPP_EXTENSION+ = .cxx

# 生成 SWIG 封闭代码(indention should be tabs for this block)
%_wrap.$(MY_SWIG_TYPE) : %.i
	$(call host-mkdir,$(MY_SWIG_OUTDIR))
	swig -java \
	$(MY_SWIG_MODE) \
	-package $(MY_SWIG_PACKAGE) \
	-outdir $(MY_SWIG_OUTDIR) \
	$<
```

2. 将 SWIG 集成到 Android.mk

为了使用这个构建系统片段，需要修改现有的 Android.mk 文件。为了运行该构建系统片段，需要在 Android.mk 文件中定义三个新变量：

- **MY_SWIG_PACKAGE:** 定义 SWIG 生成代理类的 Java 包 ，本例中为 com.apress.swig package。
- **MY_SWIG_INTERFACES:** 应该处理的 SWIG 接口文件列表，本例中为 Unix.i file。
- **MY_SWIG_MODE：** SWIG 生成封装程序的指令代码类型，为 C 或 C++，本例中为 C 代码。

打开 Project Explorer，展开根目录下的 jni 目录，在编辑器视图下打开 Android.mk。现在开始定义该项目的新变量，Android.mk 文件增加的内容如程序清单 4-9 粗体部分所示。

程序清单 4-9 在 Android.mk file 中定义 SWIG 变量

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := hello-jni
LOCAL_SRC_FILES := hello-jni.c

MY_SWIG_PACKAGE := com.apress.swig
MY_SWIG_INTERFACES := Unix.i
MY_SWIG_TYPE := c

include $(LOCAL_PATH)/my-swig-generate.mk


include $(BUILD_SHARED_LIBRARY)
```

定义了这些新变量之后，Android.mk 文件就包含了本节前面定义的 my-swig-generate.mk 构建系统片段。构建系统片段首先创建 Java 包目录，然后根据这些变量设置适当参数来调用 SWIG。由于 SWIG 生成的封装代码也应该编译到共享库里，因此上述操作应该在建立共享库之前完成。构建系统片段自动地将生成的封装文件追加到 LOCAL_SRC_FILES 变量中。

如图 4-14 所示，在主菜单选择 Project | Build All 来重新构建当前项目，Android NDK 构建日志表明 Unix_wrapper.c 封装代码将被编译到共享库中。

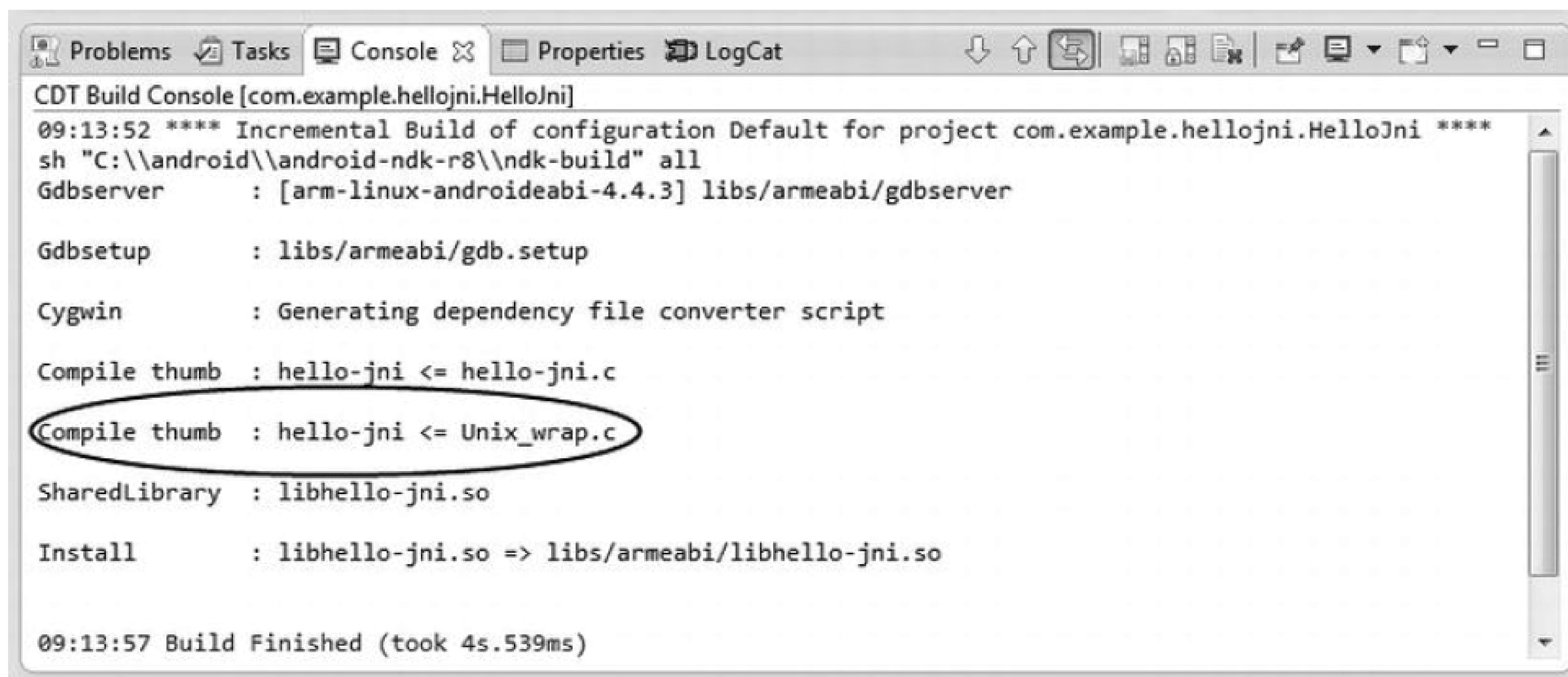


图 4-14 构建日志表明封装代码已被编译

4.3.4 更新 Activity

现在通过 Unix 代理 Java 类准确地展示了 getuid 函数。为了校验该函数，可以修改

HelloJni activity，在显示器上显示返回值。打开 Project Explorer，展开 src 目录，然后展开 com.example.hellojni Java 包，在文本编辑器中打开 HelloJni，按照程序清单 4-10 所示修改 onCreate 方法体。

程序清单 4-10　从 Unix 代理类中调用 getuid 函数

```
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);

    . . .

    TextView tv = new TextView(this);
    tv.setText("UID: " + Unix.getuid());
    setContentView(tv);
}
```

4.3.5　执行应用程序

现在应用程序准备就绪，在主菜单上选择 Run | Run 打开图 4-15 所示的应用程序，activity 将调用 getuid 函数，而且结果将显示在屏幕上。



图 4-15　activity 显示用户 ID

从示例程序中可以看到，SWIG 能够自动生成向 Java 展示原生函数所有必要的 JNI 和 Java 代码。

4.3.6　剖析生成的代码

为了能够在 Java 中访问原生函数，SWIG 生成了两个 Java 类和一个 C/C++封装代码：

- **Unix_wrap.c**：包含用于处理类型映射以及将选中的原生函数展示给 Java 的 JNI 封装函数，生成的封装函数如程序清单 4-11 所示。

程序清单 4-11　生成的封装 getuid 函数

```
SWIGEXPORT jlong JNICALL Java_com_apress_swig_UnixJNI_getuid(JNIEnv *jenv,
jclass jcls)
{
  jlong jresult = 0 ;
  uid_t result;
```

```
    (void)jenv;
    (void)jcls;
    result = (uid_t)getuid();
    jresult = (jlong)result;
    return jresult;
}
```

- **UnixJNI.java**：包含封装程序展示的所有函数的 Java 原生函数声明的中介 JNI 类，根据 Android.mk 文件的指定，它被构建在 com.apress.swig Java 包中，生成的中介 JNI 类如程序清单 4-12 所示。

程序清单 4-12　构建 JNI 中间类

```
package com.apress.swig;

public class UnixJNI {
  public final static native long getuid();
}
```

- **Unix.java**：含有所有方法以及全局变量值获取和设置的模块类，它将方法调用封装在中间 JNI 类中以实现静态类型检查。在介绍 SWIG 处理对象的方法时将会再次讨论这个问题。它也被构建在 com.apress.swig Java 包中，生成的模块类如程序清单 4-13 所示。

程序清单 4-13　生成的模块类

```
package com.apress.swig;

public class Unix {
  public static long getuid() {
    return UnixJNI.getuid();
  }
}
```

4.4　封装 C 语言代码

在前面的示例中，已经学过怎样通过 SWIG 展示函数。本节将学习如何用 SWIG 封装其他组件。需要注意的是接口文件中定义的组件只是 SWIG 用来在 Java 中展示的；除非在插入的预处理程序声明中做了声明，否则它们不会包含在生成的文件中。SWIG 假定所有被展示的组件都在代码中的某处定义过。如果这个组件没有被定义，在编译时构建会失败。

4.4.1　全局变量

尽管没有像 Java 那样的全局变量，SWIG 也支持全局变量。SWIG 在模块类里生成 getter 和 setter 方法来提供对原生全局变量的访问。为了向 Java 展示全局变量，直接将其添加到

如程序清单 4-14 所示的接口文件中。

程序清单 4-14　展示 Counter 全局变量的接口文件

```
%module Unix
. . .
/* 全局变量 counter. */
extern int counter;
```

处理接口文件时，SWIG 将生成相应的 getter 和 setter 方法，以便于在 Java 应用程序中访问全局变量，如程序清单 4-15 所示。

程序清单 4-15　计数器全局变量的 Getter and Setter 方法

```
package com.apres.swig;

public class Unix {
  ...

  public static void setCounter(int value) {
    UnixJNI.counter_set(value);
  }

  public static int getCounter() {
    return UnixJNI.counter_get();
  }
}
```

除了变量，SWIG 也提供对运行时其值不能改变的常量的支持。

4.4.2　常量

在接口文件中，常量可以用#define 或%constant 预处理指令定义，如程序清单 4-16 所示。

程序清单 4-16　定义两个常量的接口文件

```
%module Unix
. . .
/* 用 define 指令定义的常量. */
#define MAX_WIDTH 640
/* 用%constant 指令定义的常量 */
%constant int MAX_HEIGHT = 320;
```

SWIG 生成一个名为<Module>Constant 的 Java 接口，在那个接口中常量被展示为静态 final 变量，如程序清单 4-17 所示。

程序清单 4-17　展示两个常量的 UnixConstants 接口

```
package com.apress.swig;
```

```
public interface UnixConstants {
  public final static int MAX_WIDTH = UnixJNI.MAX_WIDTH_get();
  public final static int MAX_HEIGHT = UnixJNI.MAX_HEIGHT_get();
}
```

默认情况下，SWIG 生成运行库常量。通过运行库在原生代码中调用 JNI 函数完成常量初始化。可以在接口文件中使用%javaconst 预处理指令修改常量值，如程序清单 4-18 所示。

程序清单 4-18　为 MAX_WIDTH 生成编译时常数的指令

```
%module Unix
. . .
/* 用定义指令定义常量. */
%javaconst(1);
#define MAX_WIDTH 640

/* 用%constant 指令定义常量*/
%javaconst(0);
%constant int MAX_HEIGHT = 320;
```

这个预处理指令告诉 SWIG 生成一个编译时常量 MAX_WIDTH 和运行库常量 MAX_HEIGHT。现在 Java 的常量接口如程序清单 4-19 所示。

程序清单 4-19　展示编译时常量的 UnixConstants 接口

```
package com.apress.swig;

public interface UnixConstants {
  public final static int MAX_WIDTH = 640;
  public final static int MAX_HEIGHT = UnixJNI.MAX_HEIGHT_get();
}
```

特定情况下，程序开发人员希望限制对一个变量的写操作，并将它展示为 Java 只读。

4.4.3　只读变量

SWIG 提供%immutable 预处理指令来标记一个只读变量，如程序清单 4-20 所示。

程序清单 4-20　在接口文件中启用和禁用只读模式

```
%module Unix
. . .
/* 启用只读模式 */
%immutable;

/* 只读变量. */
extern int readOnly;
```

```
/* 禁用只读模式 */
%mutable;

/* 读-写变量 */
extern int readWrite;
```

不会为为只读变量生成 setter 方法，如程序清单 4-21 所示。

程序清单 4-21　不会为只读变量生成 setter 方法

```
package com.apress.swig;

public class Unix implements UnixConstants {
  . . .

  public static int getReadOnly() {
    return UnixJNI.readOnly_get();
  }

  public static void setReadWrite(int value) {
    UnixJNI.readWrite_set(value);
   }

  public static int getReadWrite() {
    return UnixJNI.readWrite_get();
  }
}
```

除了常量和只读变量，应用程序中经常使用枚举，枚举是一组命名的常量值集。

4.4.4　枚举

SWIG 可以处理命名枚举和匿名枚举。根据开发人员的不同选择或者不同的目标 Java 版本，可以通过四种不同的方法生成枚举。

1. 匿名

匿名枚举可以在接口文件中声明，如程序清单 4-22 所示。

程序清单 4-22　匿名枚举

```
%module Unix
. . .
/* 匿名枚举 */
enum { ONE = 1, TWO = 2, THREE, FOUR };
```

SWIG 在<Module>Constants Java 接口中为每个枚举生成 final 静态变量，如程序清单 4-23 所示。像常量一样，也可以生成运行库枚举。用%javaconst 预处理指令可以生成编译时枚举。

程序清单 4-23　通过常量接口展示的匿名枚举

```
package com.apress.swig;

public interface UnixConstants {
  . . .
  public final static int ONE = UnixJNI.ONE_get();
  public final static int TWO = UnixJNI.TWO_get();
  public final static int THREE = UnixJNI.THREE_get();
  public final static int FOUR = UnixJNI.FOUR_get();
}
```

2. 类型-安全

命名枚举可以在接口文件中声明，如程序清单 4-24 所示。和匿名枚举不同，命名枚举展示给 Java 的是类型-安全枚举。

程序清单 4-24　命名枚举

```
%module Unix
. . .
/* 命名枚举 */
enum Numbers { ONE = 1, TWO = 2, THREE, FOUR };
```

SWIG 用枚举名定义了一个单独的类，对应的枚举值被展示为 final 静态成员域，如程序清单 4-25 所示。

程序清单 4-25　展示为 Java 类的命名枚举

```
package com.apress.swig;

public final class Numbers {
  public final static Numbers ONE = new Numbers
    ("ONE", UnixJNI.ONE_get());
  public final static Numbers TWO = new Numbers(
    "TWO", UnixJNI.TWO_get());
  public final static Numbers THREE = new Numbers("THREE");
  public final static Numbers FOUR = new Numbers("FOUR");

  . . .
  /* Helper 方法. */
  . . .
}
```

这类枚举允许做类型检查，而且它比基于常量的方法更安全，尽管它可能不在 switch 语句中使用。

3. 类型-不安全

第三种方法是前两种方法的组合。枚举类型被封装在它所在的类中，但是枚举值以静

态 final 变量的形式展示。可以通过包含 enumtypeunsafe.swg 文件将命名枚举标记为类型-不安全枚举，如程序清单 4-26 所示。

程序清单 4-26　展示为类型不安全的命名枚举

```
%module Unix
. . .
/* 类型不安全 */
%include "enumtypeunsafe.swg"

/* 命名枚举. */
enum Numbers { ONE = 1, TWO = 2, THREE, FOUR };
```

为枚举生成的 Java 类如程序清单 4-27 所示。由于这个枚举的类型是基于常量的，所以可以用于 switch 语句中。

程序清单 4-27　展示为 Java 类的类型不安全枚举

```
package com.apress.swig;

public final class Numbers {
  public final static int ONE = UnixJNI.ONE_get();
  public final static int TWO = UnixJNI.TWO_get();
  public final static int THREE = UnixJNI.THREE_get();
  public final static int FOUR = UnixJNI.FOUR_get();
}
```

4. Java 枚举

命名枚举也可以作为恰当的 Java 枚举展示到 Java 中。这类枚举要做类型检查，而且也可用于 switch 语句中。可以通过包含 enums.swg 扩展名把命名枚举标记为 Java 枚举，如程序清单 4-28 所示。

程序清单 4-28　Java 枚举

```
%module Unix
. . .
/* Java 枚举 */
%include "enums.swg"

/* 命名枚举. */
enum Numbers { ONE = 1, TWO = 2, THREE, FOUR };
```

生成的 Java 类如程序清单 4-29 所示。

程序清单 4-29　生成的 Java 枚举类

```
package com.apress.swig;

public enum Numbers {
```

```
  ONE(UnixJNI.ONE_get()),
  TWO(UnixJNI.TWO_get()),
  THREE,
  FOUR;

  . . .
  /* Helper 方法. */
  . . .
}
```

结构体广泛应用于 C / C++应用程序中，它们将一组命名变量整合成单一的数据类型。

4.4.5 结构体

SWIG 也支持结构体。可以在接口文件中声明结构体，如程序清单 4-30 所示。

程序清单 4-30 在接口文件中声明的 Point 结构体

```
%module Unix
. . .
/* Point 结构体. */
struct Point {
  int x;
  int y;
};
```

它们和成员变量的 getters 方法及 setters 方法一起封装为 Java 类，如程序清单 4-31 所示。

程序清单 4-31 生成的 Point Java 类

```
package com.apress.swig;

public class Point {
  private long swigCPtr;
  protected boolean swigCMemOwn;

  protected Point(long cPtr, boolean cMemoryOwn) {
    swigCMemOwn = cMemoryOwn;
    swigCPtr = cPtr;
  }

  protected static long getCPtr(Point obj) {
    return (obj == null) ? 0 : obj.swigCPtr;
  }

  protected void finalize() {
    delete();
  }
```

```
    public synchronized void delete() {
      if (swigCPtr ! = 0) {
        if (swigCMemOwn) {
          swigCMemOwn = false;
          UnixJNI.delete_Point(swigCPtr);
        }
        swigCPtr = 0;
      }
    }

    public void setX(int value) {
      UnixJNI.Point_x_set(swigCPtr, this, value);
    }

    public int getX() {
      return UnixJNI.Point_x_get(swigCPtr, this);
    }

    public void setY(int value) {
      UnixJNI.Point_y_set(swigCPtr, this, value);
    }

    public int getY() {
      return UnixJNI.Point_y_get(swigCPtr, this);
    }

    public Point() {
      this(UnixJNI.new_Point(), true);
    }
  }
```

另一个广泛使用的 C/C++数据类型是指针，它是一个内存地址，其值直接指向内存中另一个地址的值。

4.4.6　指针

SWIG 也支持指针。如前面的示例程序所示，SWIG 存储的是 Java 类中实际的 C 语言结构体实例的 C 指针。SWIG 用 long 数据类型来存储指针。它通过使用 finalize 方法管理 C 语言组件的生存期，该 C 语言组件的生存期与相关 Java 类的生存期是一致的。

4.5　封装 C++代码

第 4.4 节学习了封装 C 组件的基本知识，现在来学习封装 C++代码。首先，需要修改 Android.mk 文件以使 SWIG 生成 C++代码。在 Editor 视图下打开 Android.mk 文件，并设置 MY_SWIG_TYPE 变量为 cxx，如程序清单 4-32 所示。

程序清单 4-32　Android.mk 指导 SWIG 生成 C++代码

```
MY_SWIG_PACKAGE := com.apress.swig
MY_SWIG_INTERFACES := Unix.i
MY_SWIG_TYPE := cxx
```

现在 SWIG 生成的是 C++封装器而不是 C 语言的代码。我们已经学习了函数生成，现在要重点学习传给这些函数的参数类型。

4.5.1　指针、引用和值

在 C/C++中，函数可以用许多不同的方法传递参数，比如通过指针、引用或是直接传值(如程序清单 4-33 所示)。

程序清单 4-33　带有不同参数类型的函数

```
/* 通过指针. */
void drawByPointer(struct Point* p);

/* 通过引用*/
void drawByReference(struct Point& p);

/* 通过值 */
void drawByValue(struct Point p);
```

Java 中没有这些类型，SWIG 在封装代码中把这些类型统一封装为对象实例引用，如程序清单 4-34 所示。

程序清单 4-34　在生成的 Java 类中统一方法

```
package com.apress.swig;

public class Unix implements UnixConstants {
  . . .

  public static void drawByPointer(Point p) {
    UnixJNI.drawByPointer(Point.getCPtr(p), p);
  }

  public static void drawByReference(Point p) {
    UnixJNI.drawByReference(Point.getCPtr(p), p);
  }

  public static void drawByValue(Point p) {
    UnixJNI.drawByValue(Point.getCPtr(p), p);
  }
}
```

尽管在 C 语言中声明不同，程序清单 4-35 中列出来的所有调用都是正确基于生成代

码的。

程序清单 4-35　用相同的参数类型一致地调用方法

```
Point p;
. . .
Unix.drawByPointer(p);
Unix.drawByReference(p);
Unix.drawByValue(p);
```

C/C++编程语言允许函数为它们的部分参数指定默认参数值。如果调用这些函数时省略参数，就使用默认值。

4.5.2　默认参数

尽管 Java 中不支持默认参数，SWIG 通过为每个默认参数生成附加函数来支持有默认参数的函数。带默认参数的函数可以在接口文件中声明，如程序清单 4-36 所示。

程序清单 4-36　接口文件中带默认参数的函数

```
%module Unix
. . .
/* 带默认参数的函数. */
void func(int a = 1, int b = 2, int c = 3);
```

生成的附加函数将通过模块 Java 类展示，如程序清单 4-37 所示。

程序清单 4-37　生成支持默认参数的附加函数

```
package com.apress.swig;

public class Unix {
  . . .

  public static void func(int a, int b, int c)
    UnixJNI.func__SWIG_0(a, b, c);
  }

  public static void func(int a, int b) {
    UnixJNI.func__SWIG_1(a, b);
  }

  public static void func(int a) {
    UnixJNI.func__SWIG_2(a);
  }

  public static void func() {
    UnixJNI.func__SWIG_3();
  }
}
```

函数重载允许应用程序定义多个具有相同名字但不同参数的函数。

4.5.3 重载函数

由于 Java 已经提供了对重载函数的支持，所以 SWIG 可以很轻易地支持重载函数。重载函数可以在接口文件中声明，如程序清单 4-38 所示。

程序清单 4-38 在接口文件中声明重载函数

```
%module Unix
. . .
/* 重载函数. */
void func(double d);
void func(int i);
```

SWIG 通过模块 Java 类展示重载函数，如程序清单 4-39 所示。

程序清单 4-39 通过模块 Java 类展示重载函数

```
package com.apress.swig;

public class Unix {
  . . .
  public static void func(double d) {
    UnixJNI.func__SWIG_0(d);
  }

  public static void func(int i) {
    UnixJNI.func__SWIG_1(i);
  }
}
```

SWIG 通过消除歧义的模式来解决重载函数问题，该模式按照一组类型优先规则对声明进行分级和排序。除了函数和基本数据类型，SWIG 也可以翻译 C++类。

4.5.4 类

与结构体类似，类也被封装成 Java 类。SWIG 为所有的公共类变量生成必要的 getter 和 setter 方法。类可以在接口文件中声明，如程序清单 4-40 所示。

程序清单 4-40 接口文件中的类声明

```
%module Unix
. . .
/* 类 A. */
class A {
public:
  A();
  A(int value);
```

```
  ~A();

  void print();

  int value;
private:
  void reset();
};
```

SWIG 生成相应的 Java 类，如程序清单 4-41 所示。值成员变量是公共的，SWIG 自动生成相应的 getter 和 setter 方法。因为 reset 方法在类声明中被定义为私有的，因此它没有被展示到 Java 中。

程序清单 4-41　C/C++展示到 Java

```
package com.apress.swig;

public class A {
  private long swigCPtr;
  protected boolean swigCMemOwn;
  protected A(long cPtr, boolean cMemoryOwn) {
    swigCMemOwn = cMemoryOwn;
    swigCPtr = cPtr;
  }

  protected static long getCPtr(A obj) {
    return (obj == null) ? 0 : obj.swigCPtr;
  }

  protected void finalize() {
    delete();
  }

  public synchronized void delete() {
    if (swigCPtr ! = 0) {
      if (swigCMemOwn) {
        swigCMemOwn = false;
        UnixJNI.delete_A(swigCPtr);
      }
      swigCPtr = 0;
    }
  }

  public A() {
    this(UnixJNI.new_A__SWIG_0(), true);
  }

  public A(int value) {
    this(UnixJNI.new_A__SWIG_1(value), true);
  }
```

```
  public void print() {
    UnixJNI.A_print(swigCPtr, this);
  }

   public void setValue(int value) {
    UnixJNI.A_value_set(swigCPtr, this, value);
  }

  public int getValue() {
    return UnixJNI.A_value_get(swigCPtr, this);
  }
}
```

SWIG 也支持继承。被封装到有层次关系的 Java 类中的这些类在 C++中也具有相同的继承关系。由于 Java 不支持多继承，任何多继承的 C++类在代码生成阶段都会产生错误。

4.6 异常处理

在原生代码中，C/C++函数能够抛出异常或者返回错误代码。SWIG 允许开发人员通过使用%exception 预处理指令将 C/C++异常和错误转换成 Java 异常，进而将异常处理代码引入生成的封装代码中。异常处理代码可以在接口文件中定义，如程序清单 4-42 所示。异常处理代码应该在实际函数声明前定义。

程序清单 4-42　getuid 函数的异常处理代码

```
/* getuid 函数的异常处理 */
%exception getuid {
  $action
  if (!result) {
    jclass clazz = jenv->FindClass("java/lang/OutOfMemoryError");
    jenv->ThrowNew(clazz, "Out of Memory");
    return $null;
  }
}

/* 请求 SWIG 封装 getuid 函数 */
extern uid_t getuid(void);
```

与程序清单 4-11 相比，生成的 getuid 函数封装器代码如程序清单 4-43 所示。

程序清单 4-43　带异常处理的封装代码

```
SWIGEXPORT jlong JNICALL Java_com_apress_swig_UnixJNI_getuid(JNIEnv *jenv,
jclass jcls) {
  jlong jresult = 0 ;
  uid_t result;
```

```
  (void)jenv;
  (void)jcls;
  {
    result = (uid_t)getuid();
    if (!result) {
      jclass clazz = jenv->FindClass("java/lang/OutOfMemoryError");
      jenv- > ThrowNew(clazz, "Out of Memory");
      return 0;
    }
  }
  jresult = (jlong)result;
  return jresult;
}
```

由于代码抛出了一个运行库异常，生成的 Java 代码没有改变。如果抛出一个检查的异常，SWIG 可以通过%javaexception 预处理指令做出反应并生成相应的 Java 方法，如程序清单 4-44 所示。

程序清单 4-44　告诉 SWIG 可能抛出一个检查异常

```
/* getuid 的异常处理*/
%javaexception("java.lang.IllegalAccessException") getuid {
  $action
  if (!result) {
    jclass clazz = jenv->FindClass("java/lang/IllegalAccessException");
    jenv->ThrowNew(clazz, "Illegal Access");
    return $null;
  }
}
```

现在生成的 Java 方法签名表明可能抛出检查异常，如程序清单 4-45 所示。

程序清单 4-45　表明抛出异常的 Java 类

```
package com.apress.swig;

public class Unix {
  public static long getuid() throws java.lang.IllegalAccessException {
    return UnixJNI.getuid();
  }
}
```

4.7　内存管理

SWIG 生成的每一个代理类都包含一个名为 swigCMemOwn 的所有权标志。这个标志指定谁负责清理底层 C/C++组件。如果底层组件属于这个代理类，回收机制通过 Java 类的 finalize 方法来释放内存，也可以不用等待回收机制而直接调用 Java 类的删除方法来释放内存。在运行期间，Java 类可以通过 swigReleaseOwnership 释放 C/C++底层组件的内存所

有权、通过 swigTakeOwnership 方法取得 C/C++底层组件的内存所有权。

4.8 从原生代码中调用 Java

目前为止，学习了各种从 Java 中调用 C/C++ 代码的方法。在某些情况下，也需要在 C/C++代码反过来调用 Java 代码，例如回调。SWIG 也支持使用虚拟方法在 C/C++ 代码调用 Java 代码。

4.8.1 异步通信

为了演示这一流程，需要把 getuid 函数调用封装到一个 C/C++类并通过一个回调返回它的结果，从而将 getuid 函数调用转换成异步模式。为了这个实验，可以将类声明和类定义放到 SWIG 接口文件中，如程序清单 4-46 所示。

程序清单 4-46 AsyncUidProvider 类的定义和声明

```
%module Unix
. . .
%{
/* Asynchornous user ID 提供者 */
class AsyncUidProvider {
public:
  AsyncUidProvider() {
  }

  virtual ~ AsyncUidProvider() {
  }

  void get() {
    onUid(getuid());
  }

  virtual void onUid(uid_t uid) {
  }
};
%}

/* Asynchornous user ID provider. */
class AsyncUidProvider {
public:
  AsyncUidProvider();
  virtual ~ AsyncUidProvider();

  void get();
  virtual void onUid(uid_t uid);
};
```

4.8.2　启用 Directors

SWIG 用 directors 特征提供对交叉语言多态性的支持。默认情况下，该特性是禁用的，为了启用这一特征，需要修改%module 预处理指令让它包含 directors 标志。在启用 directors 功能之后，要用%feature 预处理指令将特征应用于 AsyncUidProvider 中，两处修改请见程序清单 4-47。

程序清单 4-47　开启 Directors 扩展并应用该特征

```
/* 模块名为 Unix. */
%module(directors = 1) Unix

/* 启用 AsyncUidProvider 的 directors */
%feature("director") AsyncUidProvider;
```

为了能够从 C/C++代码中调用 Java 代码，directors 扩展依赖于编译器的运行库类型信息特征(RTTI，Run-Time Type Information)。

4.8.3　启用 RTTI

默认情况下，Android NDK 构建系统中 RTTI 是关闭的。为了启用它，按照程序清单 4-48 所示的内容修改 Android.mk 文件。

程序清单 4-48　启用 Android.mk 文件中的 RTTI

```
# 启用 RTTI
LOCAL_CPP_FEATURES + = rtti
```

现在原生代码部分已经准备好了，在顶部菜单中单击 Project | Build All 重新构建当前项目。

4.8.4　重写回调方法

在 Java 端，需要对展示的 AsyncUidProvider 类进行扩展，并重写 onUid 方法来接收 getuid 函数调用的返回值，如程序清单 4-49 所示。

程序清单 4-49　在 Java 中扩展 AsyncUidProvider

```
package com.example.hellojni;

import android.widget.TextView;

import com.apress.swig.AsyncUidProvider;

public class UidHandler extends AsyncUidProvider {
  private final TextView textView;
```

```
    UidHandler(TextView textView) {
      this.textView = textView;
    }

    @Override
    public void onUid(long uid) {
      textView.setText("UID: " + uid);
    }
  }
```

4.8.5 更新 HelloJni Activity

最后一步，修改 HelloJni Activity 以使用 UidHandler 类。onCreate 方法中需要修改的内容如程序清单 4-50 所示。

程序清单 4-50 用新的 UidHandler 类修改 onCreate 方法

```
@Override
public void onCreate(Bundle savedInstanceState)
{
  . . .

  TextView  tv = new TextView(this);
  setContentView(tv);

  UidHandler uidHandler = new UidHandler(tv);
  uidHandler.get();
}
```

在主菜单中单击 New | File 来打开这个应用程序，调用 AsnycUidProvider 类的 get 方法时，C/C++代码用 getuid 函数调用的返回结果回调 Java 并且显示调用结果。

4.9 小结

正如你所见，SWIG 通过生成必要的 JNI 封装代码简化了原生代码与 Java 层之间的连接过程。尽管这个过程对开发人员是高度透明的，SWIG 仍然支持让开发人员对生成的代码进行扩展以满足他们的个性化需求的必要功能。关于 SWIG 的更多资料请登录 http://swig.org/Doc2.0/index.html 查找 SWIG 文档。在下面的章节中将继续学习 SWIG 的其他特征并且经常会使用 SWIG。

第 5 章

日志、调试及故障处理

第 4 章学习了 Android NDK 构建系统以及用 JNI 技术连接原生代码与 Java 应用程序的方法。毋庸置疑，学习新平台上的应用程序开发需要花很多的时间做实验。在用提供的原生 API 做实验之前，掌握 Android 平台的故障排除技术是至关重要的，因为它可以帮助我们快速发现问题，从而加快学习进度。因为 Android 应用程序的开发和执行在不同的机器上进行，因此不能直接使用现有的故障排除技术。本章将学习日志、调试和故障排除工具及技术，主要内容包括：

- Android 日志框架概述
- 用 Eclipse 和命令行方式调试原生代码
- 在机器崩溃时分析栈跟踪
- 用 CheckJNI 模式尽早发现故障
- 用 libc 和 Valgrind 处理内存相关故障
- 用 strace 监控原生代码执行情况

5.1 日志

日志是故障处理中最重要的部分，但是它难以实现，特别是在那些使用两个不同的机器进行开发和执行的移动平台上。Android 有一个扩展日志框架，用于对系统范围内 Android 系统本身的信息及应用程序的信息集中做日志。它还提供了一组用户级应用程序以查看和过滤这些日志，如 logcat 和 Dalvik 调试监视服务器(DDMS，Dalvik Debug Monitor Server)工具。

5.1.1 框架

Android 日志框架是名字为 logger 的内核模块。随时随地地对平台上的任何信息进行日志会产生大量信息，从而使得查看和分析这些日志变得非常困难。为了简化这个过程，

Android 日志框架把日志消息分成 4 个日志缓冲区：

- **Main**：主要应用程序的日志信息
- **Event**：系统事件
- **Radio**：Radio 相关的日志信息
- **System**：调试时产生的低级系统调试信息

这 4 个缓冲区以伪设备的形式保存在/dev/log 系统目录下。因为移动平台上的 I/O 操作代价很大，所以日志信息要保存在内存中，而不能保存在永久性存储器(例如磁盘)中。为了有效控制对存储日志信息的内存空间的利用，logger 模块把日志信息放在固定大小的缓冲区。Main、radio 和 system 日志以自由文本的格式保存在 64KB 的日志缓存区中。事件日志信息带有额外的二进制形式信息，因此保存在 256KB 的日志缓存区中。

5.1.2 原生日志 API

开发者不希望直接与 logger 内核模块进行交互。Android 运行库系统提供了一组 API 调用以便于 Java 代码和原生代码向 logger 内核模块发送日志信息。通过 android/log.h 头文件来展示原生代码的日志 API。为了使用日志函数，原生代码需要先包含该头文件。

```
#include <android.h>
```

除了要包含合适的头文件，还需要动态修改 Android.mk 文件从而将原生模块与日志库进行链接，可以通过使用构建系统变量 LOCAL_LDLIBS 完成该操作，如程序清单 5-1 所示。该构建系统变量必须被放在共享库构建片段的 include 语句之前；否则它将不起作用。

程序清单 5-1　动态链接原生模块与日志库

```
LOCAL_MODULE:=hello-jni
…
LOCAL_LALIBS+=-llog
…
Include $(BUILD_SHARED_LIBRARY)
```

1. 日志消息

通过日志 API 发送给 logger 模块的每个日志条目都具有以下字段：

- **Priority**：取值分别为 verbose、debug、info、warning、error 和 fatal，表示日志信息的重要程度。支持的日志优先级在 android/log.h 头文件中声明，如程序清单 5-2 所示。

程序清单 5-2　支持的日志优先级

```
typedef enum android_LogPriority {
  ...
  ANDROID_LOG_VERBOSE,
  ANDROID_LOG_DEBUG,
  ANDROID_LOG_INFO,
  ANDROID_LOG_WARN,
```

```
    ANDROID_LOG_ERROR,
    ANDROID_LOG_FATAL,
    ...
} android_LogPriority;
```

- **Tag：**标识产生日志信息的组件，Logcat 和 DDMS 工具可以基于这个标签值过滤日志信息。标签值应该尽可能小。
- **Message：**用于存放实际日志信息。在每一个日志消息后自动追加一个换行符，因为循环的日志缓存区很小，因此强烈建议应用程序的日志信息大小尽量保持合适。

2. 日志函数

android/log.h 头文件也声明了一系列函数，这些函数主要用于原生代码生成日志消息。

- _android_log_write：可用于生成一个简单的字符串作为日志信息。如程序清单 5-3 所示，它包括日志优先级、日志标签和日志消息。

程序清单 5-3　生成简单的日志消息

```
_android_log_write(ANDROID_LOG_WARN,"hello-jni","warning log.");
```

- _android_log_print：可用于生成一个格式化字符串作为日志消息。它包括日志优先级、日志标签、字符串格式和格式中指定个数的其他参数，如程序清单 5-4 所示。请查阅 ANSI C printf 文档了解格式化字符串的语法。

程序清单 5-4　生成格式化的日志消息

```
_android_log_print(ANDROID_LOG_ERROR,"hello-jni",
    "Failed with errno %d",erron);
```

- _android_log_vprint：除了参数传递方式外，其他功能与_android_log_print 函数完全相同，_android_log_vprint 函数用 va_list 传递附加参数，而_android_log_print 函数中以连续参数的方式改为传递参数。如果想要调用日志函数时传递给当前函数的参数个数动态变化时，该函数的优势就会体现出来，如程序清单 5-5 所示。

程序清单 5-5　传递的参数个数变化时生成日志消息

```
void log_verbose(const char* format,…)
{
   va_list args;

   va_start(args,format);
   _android_log_vprint(ANDORID_LOG_VERBOSS,"hello-jni",format,args);
   va_end(args);
}
…
void example()
{
    log_verbose("Errno is now  %d",errno);
}
```

● _android_log_assert：用于记录断言失败。与其他的日志函数比较，它不包括日志优先级，但总是将所有的日志记录为 fatal，如程序清单 5-6 所示。如果附加了调试器，它也 SIGTRAP 当前进程以通过 debugger 进一步检查。

程序清单 5-6　生成断言失败日志

```
if(0!=erron)
{
   _android_log_assert("0!=errno","hello-jni","There is an error.");
}
```

5.1.3　受控制的日志

与 Java 中的对应部分一样，原生的日志 API 只允许用户向日志内核模块发出日志消息，在现实生活中，在发行和调试构建时你既不会用断言也不会以相同的间隔做日志。遗憾的是，Android 日志 API 不提供任何基于优先级的日志消息压缩机制。它没有 Log4J 或 Log4CXX 等其他日志框架那样先进。Android 日志框架假设你将在新发布的版本中除去不必要的日志信息。尽管在 Java 应用程序中使用 Proguard 很容易做到这一点，但是在原生代码中却难以实现。

1. 日志包装器

本节介绍关于上述问题的基于预处理器的解决方案。为此我们修改以前导入的 hello-jni 原生项目，打开 Eclipse，在 Project Explorer 菜单项上右击 jni 子目录。在上下文菜单上，选择 New Header File 打开创建头文件对话框。将头文件名字设置为 my_log.h，单击 Finish 按钮继续。my_log.h 头文件的内容显示在程序清单 5-7 中。

程序清单 5-7　my_log.h 头文件的内容

```
#pragma once
/**
* NDK 的基本日志框架.
*
* @作者 Onur Cinar
*/

#include <android/log.h>

#define MY_LOG_LEVEL_VERBOSE 1
#define MY_LOG_LEVEL_DEBUG 2
#define MY_LOG_LEVEL_INFO 3
#define MY_LOG_LEVEL_WARNING 4
#define MY_LOG_LEVEL_ERROR 5
#define MY_LOG_LEVEL_FATAL 6
#define MY_LOG_LEVEL_SILENT 7

#ifndef MY_LOG_TAG
```

```
#    define MY_LOG_TAG __FILE__
#endif

#ifndef MY_LOG_LEVEL
#    define MY_LOG_LEVEL MY_LOG_LEVEL_VERBOSE
#endif

#define MY_LOG_NOOP (void) 0

#define MY_LOG_PRINT(level,fmt,...) \
        __android_log_print(level, MY_LOG_TAG, "(%s:%u) %s: " fmt, \
                __FILE__, __LINE__, __PRETTY_FUNCTION__, ##__VA_ARGS__)

#if MY_LOG_LEVEL_VERBOSE >= MY_LOG_LEVEL
#    define MY_LOG_VERBOSE(fmt,...) \
        MY_LOG_PRINT(ANDROID_LOG_VERBOSE, fmt, ##__VA_ARGS__)
#else
#   define MY_LOG_VERBOSE(...) MY_LOG_NOOP
#endif

#if MY_LOG_LEVEL_DEBUG >= MY_LOG_LEVEL
#    define MY_LOG_DEBUG(fmt,...) \
        MY_LOG_PRINT(ANDROID_LOG_DEBUG, fmt, ##__VA_ARGS__)
#else
#   define MY_LOG_DEBUG(...) MY_LOG_NOOP
#endif

#if MY_LOG_LEVEL_INFO >= MY_LOG_LEVEL
#   define MY_LOG_INFO(fmt,...) \
        MY_LOG_PRINT(ANDROID_LOG_INFO, fmt, ##__VA_ARGS__)
#else
#   define MY_LOG_INFO(...) MY_LOG_NOOP
#endif

#if MY_LOG_LEVEL_WARNING >= MY_LOG_LEVEL
#   define MY_LOG_WARNING(fmt,...) \
        MY_LOG_PRINT(ANDROID_LOG_WARN, fmt, ##__VA_ARGS__)
#else
#   define MY_LOG_WARNING(...) MY_LOG_NOOP
#endif

#if MY_LOG_LEVEL_ERROR >= MY_LOG_LEVEL
#    define MY_LOG_ERROR(fmt,...) \
        MY_LOG_PRINT(ANDROID_LOG_ERROR, fmt, ##__VA_ARGS__)
#else
#    define MY_LOG_ERROR(...) MY_LOG_NOOP
#endif

#if MY_LOG_LEVEL_FATAL >= MY_LOG_LEVEL
```

```
#    define MY_LOG_FATAL(fmt,...) \
         MY_LOG_PRINT(ANDROID_LOG_FATAL, fmt, ##__VA_ARGS__)
#else
#    define MY_LOG_FATAL(...) MY_LOG_NOOP
#endif

#if MY_LOG_LEVEL_FATAL >= MY_LOG_LEVEL
#    define MY_LOG_ASSERT(expression, fmt, ...) \
        if (!(expression)) \
        { \
            __android_log_assert(#expression, MY_LOG_TAG, \
                fmt, ##__VA_ARGS__); \
        }
#else
#   define MY_LOG_ASSERT(...) MY_LOG_NOOP
#endif
```

my_log.h 头文件通过一系列预处理程序指令为原生代码定义了一个基本的日志框架。这些预处理程序指令包括 Android 日志函数并允许它们在编译时被触发。

2. 增加日志

现在可以将日志声明加入到原生代码中，在 Project Explorer 视图中双击 hello-jni.c 源文件，并在 Editor 视图中打开该文件。为了使用基本的日志框架，需要先包含 my_log.h 头文件。不需要包含 android/log.h，因为 my_log.h 头文件中已经包含了 android/log.h 的内容。

```
#include "my_log.h"
```

现在可以在原生函数中添加日志声明语句，如程序清单 5-8 所示。

程序清单 5-8　在原生函数中添加日志声明语句

```
jstring
Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env,jobject
thiz )
{
    MY_LOG_VERBOSE("The stringFromJNI is called.");

    MY_LOG_DEBUG("env=%p thiz=%p", env, thiz);

    MY_LOG_ASSERT(0 != env, "JNIEnv cannot be NULL.");

    MY_LOG_INFO("Returning a new string.");

    return (*env)->NewStringUTF(env, "Hello from JNI !");
}
```

3. 更新 Android.mk

现在可以更新 Android.mk 文件来调整基本的日志框架。在 Project Explorer 视图中双击 Android.mk 源文件以在 Editor 视图中打开该文件。

(1) 日志标签

正如上文所述，每一个日志消息都包括一个标识日志来源的标签。模块中的日志标签可以在 Android.mk 中定义，如程序清单 5-9 所示。

程序清单 5-9　通过 MY_LOG_TAG 构建变量定义日志标签

```
LOCAL_MODULE := hello-jni
...
# 定义日志标签
MY_LOG_TAG := \"hello-jni\"
```

(2) 日志等级

基本日志框架的主要优点是可以定义日志等级。因为发布版本和调试版本中的日志粒度不同，可以通过修改 Android.mk 文件从而为发布版本和调试版本定义不同的日志等级，如程序清单 5-10 所示。

程序清单 5-10　定义默认的日志等级

```
LOCAL_MODULE := hello-jni
...
# 定义日志标签
MY_LOG_TAG := \"hello-jni\"

# 定义基于构建类型的默认日志等级
ifeq ($(APP_OPTIM),release)
  MY_LOG_LEVEL := MY_LOG_LEVEL_ERROR
else
  MY_LOG_LEVEL := MY_LOG_LEVEL_VERBOSE
endif
```

第 2 章曾提到，APP_OTIM 构建系统变量指定了构建类型是发布还是调试。基于变量 APP_OPTIM 的值，将 MY_LOG_LEVEL 的值设置成匹配的日志等级。

(3) 应用日志配置

在定义 MY_LOG_TAG 和 MY_LOG_LEVEL 构建系统变量时，可以将日志系统配置应用在模块中，如程序清单 5-11 所示。

程序清单 5-11　将日志系统配置应用在模块中

```
LOCAL_MODULE := hello-jni
...
# 定义日志标签
MY_LOG_TAG := hello-jni
```

```
#定义基于构建类型的默认日志等级
ifeq ($(APP_OPTIM),release)
  MY_LOG_LEVEL := MY_LOG_LEVEL_ERROR
else
  MY_LOG_LEVEL := MY_LOG_LEVEL_VERBOSE
endif

# 追加编译标记
LOCAL_CFLAGS += -DMY_LOG_TAG=$(MY_LOG_TAG)
LOCAL_CFLAGS += -DMY_LOG_LEVEL=$(MY_LOG_LEVEL)

# 动态链接日志库
LOCAL_LDLIBS += -llog
```

4. 通过 Logcat 查看日志消息

在执行 hello.jni 应用程序时，可以通过 Logcat 视图查看日志消息，如图 5-1 所示。

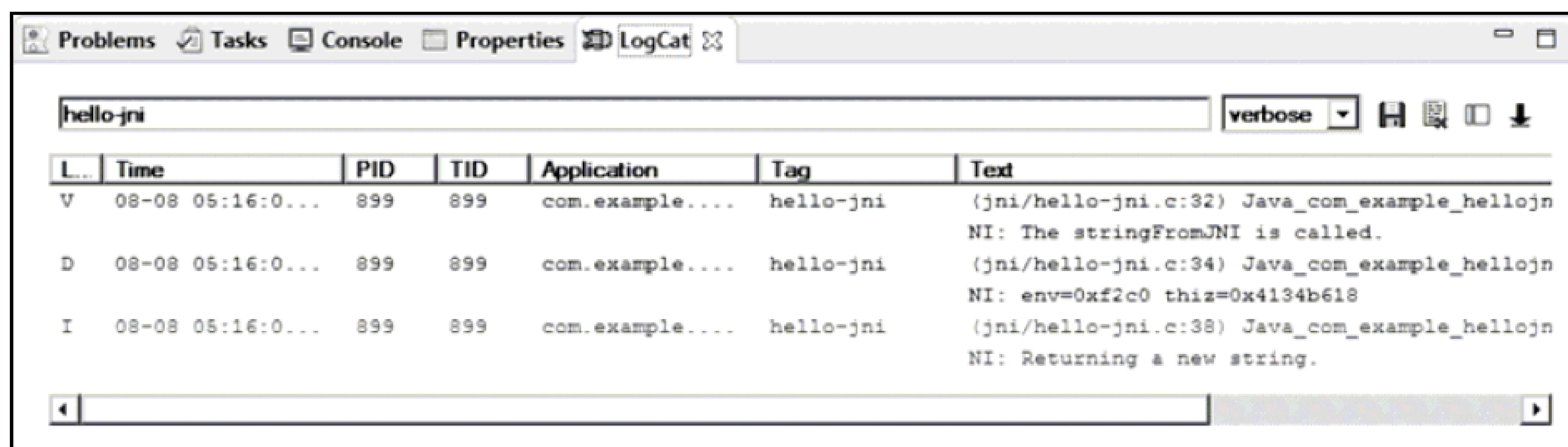


图 5-1　原生代码中的日志消息

5.1.4　控制台日志

当将第三方库文件或早期版本中的模块集成到 Android 应用程序项目中时，不可能将它们的日志机制改为 Android 特定的日志。大多数的日志机制或将日志消息写入文件或者直接写入控制台。

默认情况下，控制台文件描述符——STDOUT 和 STDERR 在 Android 平台上是不可见的。要想将这些日志消息重定向到 Android 系统日志中，需要打开一个命令提示符或一个终端窗口，并且执行程序清单 5-12 所示的 ADB 命令。

程序清单 5-12　将控制台日志重定向到 Android 系统日志中

```
adb shell stop
adb shell setprop log.redirect-stdio true
adb shell start
```

在重新启动应用程序时，用 Logcat 视图可以看到控制台日志信息，如图 5-2 所示。

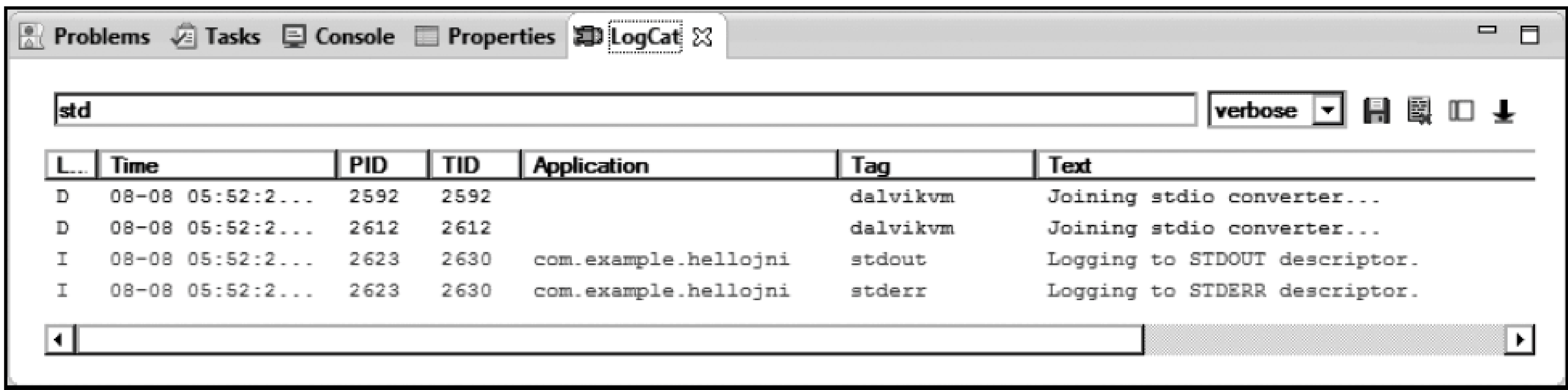


图 5-2　自 STDOUT 和 STDERR 描述符的日志消息重定向

设备重启前系统将一直保存此设置。如果想将此设置为默认值，需要将上述命令添加到设备或模拟器的/data/local.prop 文件。

5.2　调试

日志允许正在运行的应用程序输出消息以显示程序当前的状态。当进行故障检测时，从相关部分的代码中产生的日志消息的粒度是远远不够的，需要将新的日志植入代码中以显示更多关于它当前状态的信息，但是这显然使得故障诊断速度大大降低。用调试器去观察应用程序的状态是最方便的故障检测方法。Android NDK 支持通过 GNU 调试器(GDB)来调试原生代码。

5.2.1　预备知识

为了调试原生代码，必须满足下列条件：

- 可以在命令行方式下用 ndk-build 命令或在 Eclipse IDE 环境下用 Android 开发工具对原生代码进行编译。在构建过程中，NDK 构建系统生成一组文件以便于进行远程调试。
- 通过设置 AndroidManifest.xml 文件中的应用程序标签属性 android:debuggable 可以将应用程序设置成可调试的，如程序清单 5-13 所示。

程序清单 5-13　将应用程序声明为可调试的

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.example.hellojni"
      android:versionCode="1"
      android:versionName="1.0">
    ...
    <application android:label="@string/app_name"
                 android:debuggable="true">
    ...
    </application>
</manifest>
```

- 设备和仿真器需要运行 Android 2.2 或更高的版本。早期版本不支持原生代码调试。

nkd-gdb 脚本处理许多错误情况并输出大量错误信息以让用户知道哪些条件没有满足。

5.2.2 调试会话建立

ndk-gdb 脚本代表开发人员设置调试会话，但同时知道调试会话建立期间事件的发生序列，这些序列有助于理解在 Android 平台上调试原生代码的警告信息。在调试会话建立阶段事件发生的完整序列如图 5-3 所示。

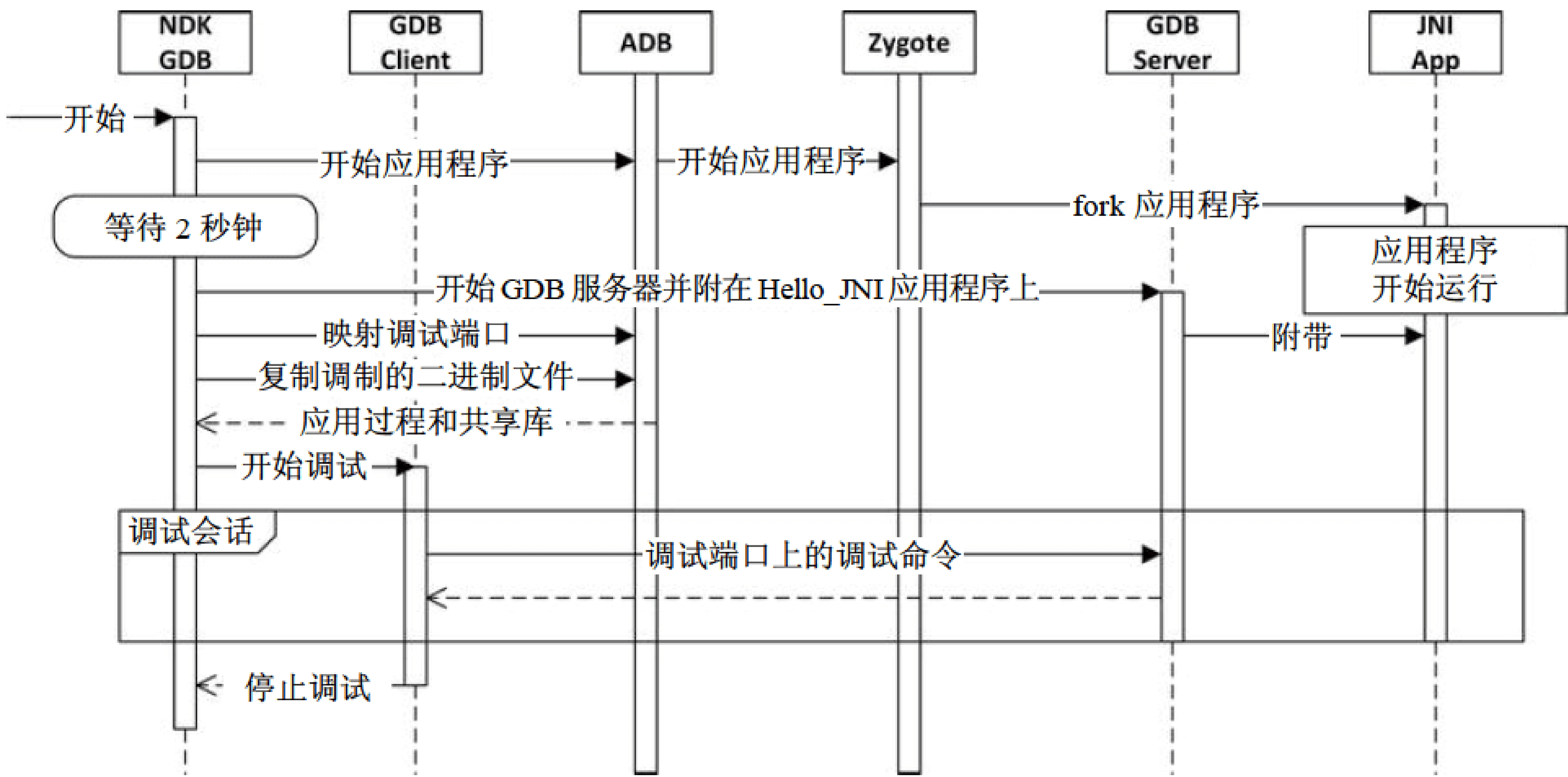


图 5-3 调试会话建立序列图

ndk-gdb 脚本用应用程序管理器的 ADB 打开目标应用程序，应用程序管理器只是将请求转给 Zygote 进程。

Zygote 也被称为“app 进程”，它是 Android 系统引导时启动的核心进程之一，它在 Android 平台扮演的角色是启动 Dalvik 虚拟机并初始化所有 Android 核心服务。作为一个移动操作系统，为了提供一个快速响应的用户体验，Android 需要确保应用程序的启动时间尽量短。为此 Zygote 不是从零开始为应用程序启动一个新进程，而是只使用 fork 这一系统调用。在计算时，fork 是克隆一个现有进程的操作。尽管两个进程都独立运行，但是事实上，新的进程复制了父进程的所有内存片段。

此时，应用程序已经启动起来了而且开始执行代码。正如你看到的，此时调试会话仍然没有建立。

注意：

基于 Zygote 的工作方式，GDB 不能启动应用程序，但是它附加于一个已经运行的应用程序进程上。如果想在 GDB 附加之前阻止应用程序执行代码，需要用 Java Debugger 在代码的合适位置设置一个断点。

在获得应用程序的进程 ID 时，ndk-gdb 脚本在 Android 平台上启动 GDB 服务器并将

其附于运行的应用程序上。ndk-gdb 脚本用 ADB 配置转发端口从而可以在主机上访问 GDB 服务器。然后，在启动 GDB 客户端之前，为 Zygote 将二进制文件和共享库复制到主机上。在二进制文件复制之后，ndk-gdb 脚本启动 GDB 客户端，同时调试会话被激活。然后便可以开始调试应用程序。

5.2.3　建立调试示例

为了学习原生代码的调试操作，需要用到 hello-jni 示例项目。为了简化调试过程，需要对 HelloJni Activity 的 onCreate 方法作一个小小的修改。如前所述，用 Eclipse 的 Editor View 窗口打开 HelloJni 活动。如程序清单 5-14 所示修改 onCreate 方法。

程序清单 5-14　修改 onCreate 方法以延长原生调用时间

```
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);

    Button button = new Button(this);
    button.setText("Call Native");
    button.setOnClickListener(new OnClickListener() {
        public void onClick(View button) {
           ((Button) button).setText(stringFromJNI());
        }
    });

    setContentView(button);
}
```

在菜单栏中，选择 Source | Organize Imports 从而让 Eclipse 向源文件中增加必要的导入语句。Eclipse 将提供多种可供选择的 OnClickListener 类导入选项。选择 android.view.View.OnClickListener 进行处理。修改的 onCreate 方法将一个按钮放在展示界面，单击该按钮将开始原生调用，这将确保调试会话正确建立之后启动原生调用。

5.2.4　启动调试器

可以使用命令行方式或者用 Eclipse 进行原生代码的调试，本节将讲解这两种方式。

1. 为 Windows 用户进行修复

Windows 平台上的 Android NDK 中有一个众所周知的 bug，它阻止 GDB 对二进制文件的正确定位。ndk-gdb 脚本用 GDB 脚本文件配置 GDB Client。在 Windows 平台上，这个脚本文件生成时带有额外的托架返回，从而导致这个问题。

为了修复上述问题，在 Eclipse 的 Editor 视图中打开<ANDROID_NDK_HOME>/ndk-gdb 脚本。将光标定位在文件的结尾，追加程序清单 5-15 所示的修复内容。

程序清单 5-15　修复 GDB 设置脚本产生器

```
# 修复行尾.
sed -i 's/\r\r//' 'native_path $GDBSETUP'

$GDBCLIENT -x 'native_path $GDBSETUP'
```

2. 使用 Eclipse

像运行应用程序一样，为了建立调试会话，Eclipse 需要定义调试配置。

(1) 在菜单栏中选择 Run|Debug Configurations 打开 Debug Configurations 对话框，如图 5-4 所示。

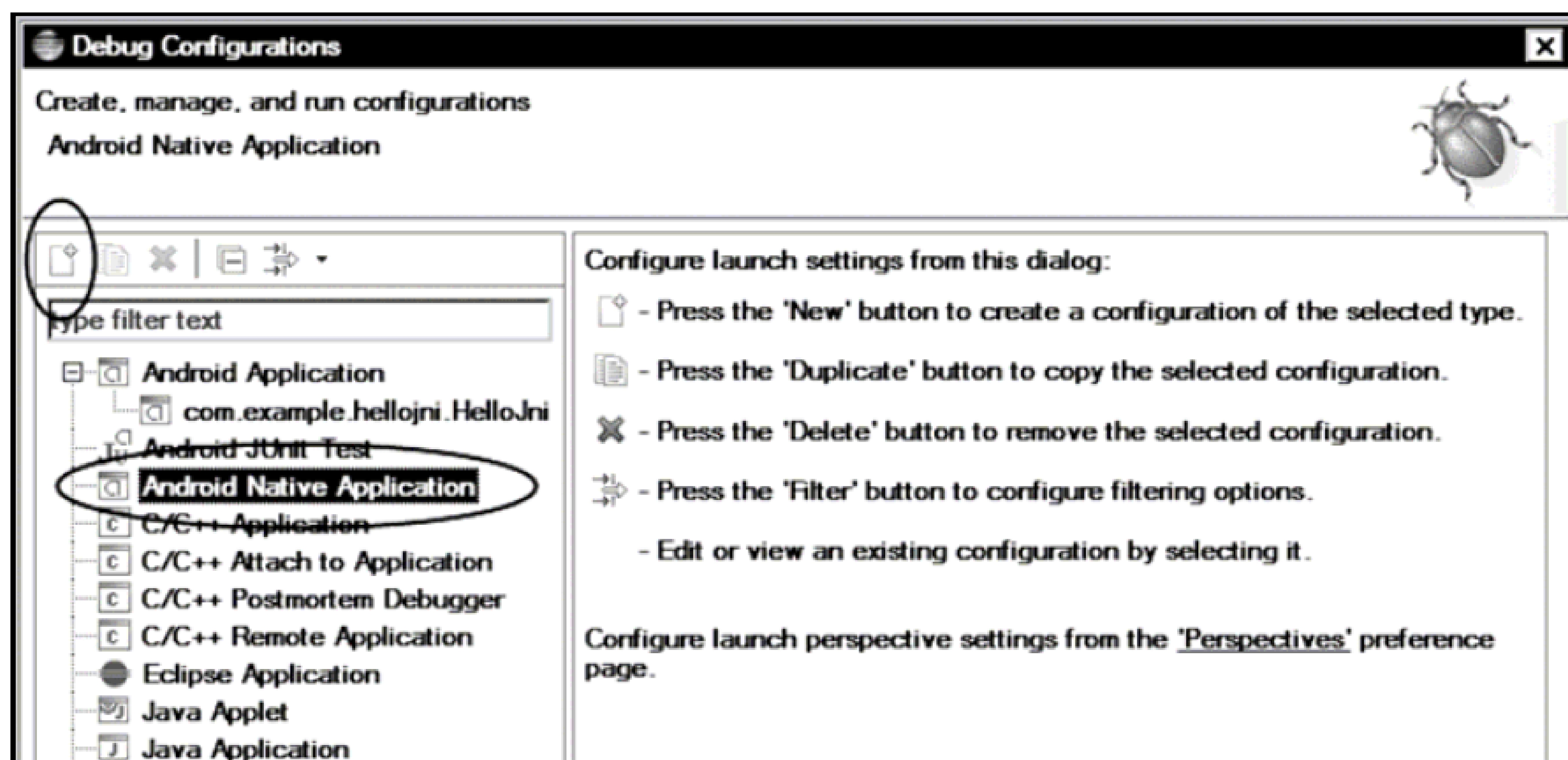


图 5-4　新建 Android 原生应用配置

(2) 在左侧的窗格中，选择 Android Native Application。

(3) 在会话工具栏单击 new configuration 图标。

(4) 如图 5-5 所示，在右侧的窗格中，单击 Browse 按钮选择当前项目。

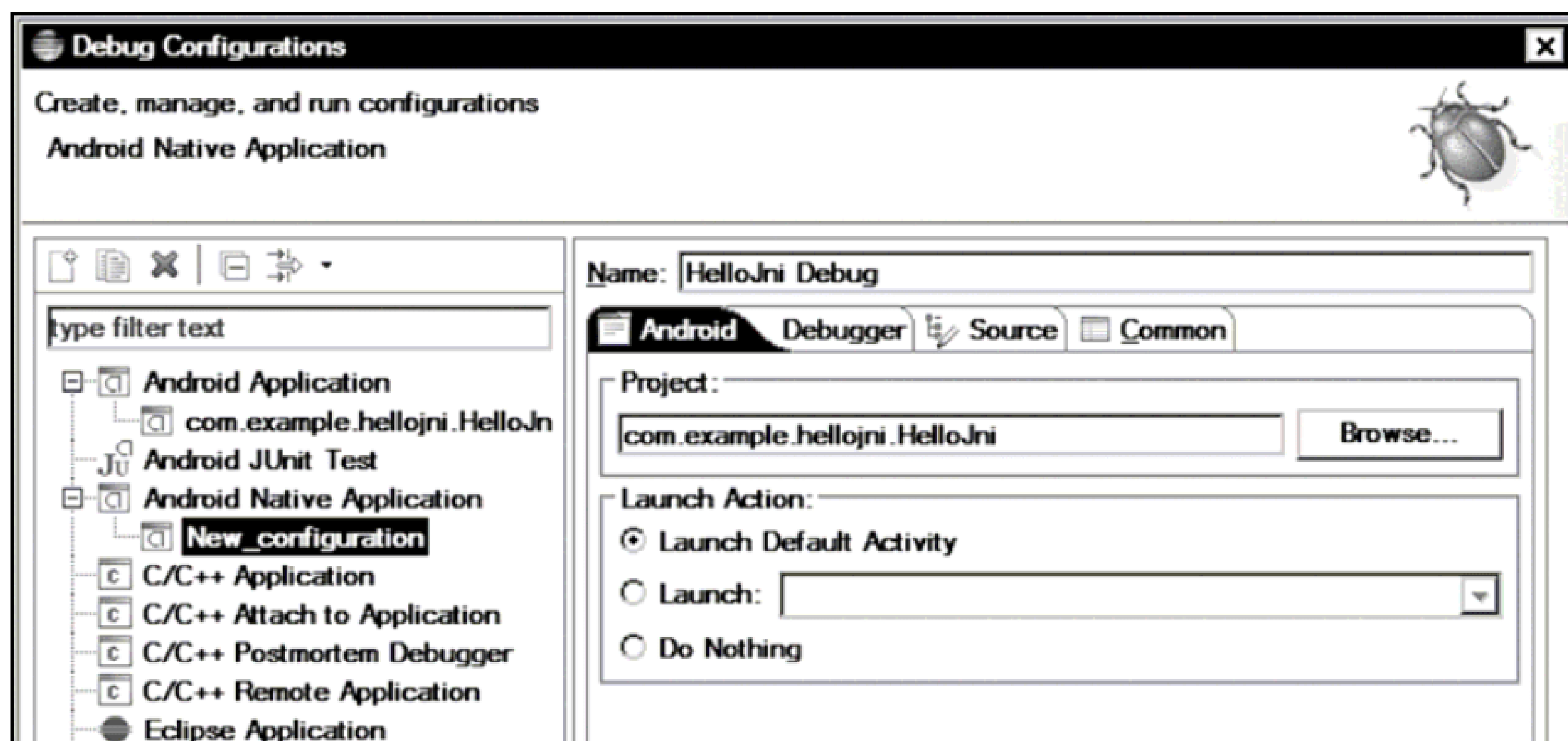


图 5-5　定义原生调试配置

(5) 单击 Apply 按钮来保存调试配置。

(6) 关闭调试配置对话框，回到 Eclipse 工作台。

现在请在原生代码中放置一个断点来终止调试进程，为此需要以下步骤：

(1) 如前所述，在 Editor 视图中打开 hello.jni.c 源文件。

(2) 进入原生函数，右击 Editor 视图左侧的标记区。

(3) 如图 5-6 所示，在上下文菜单中选择 Toggle Breakpoint 选项以放置一个断点，此时一个蓝色的圆点将会被放置在标记区，表示它是一个断点。

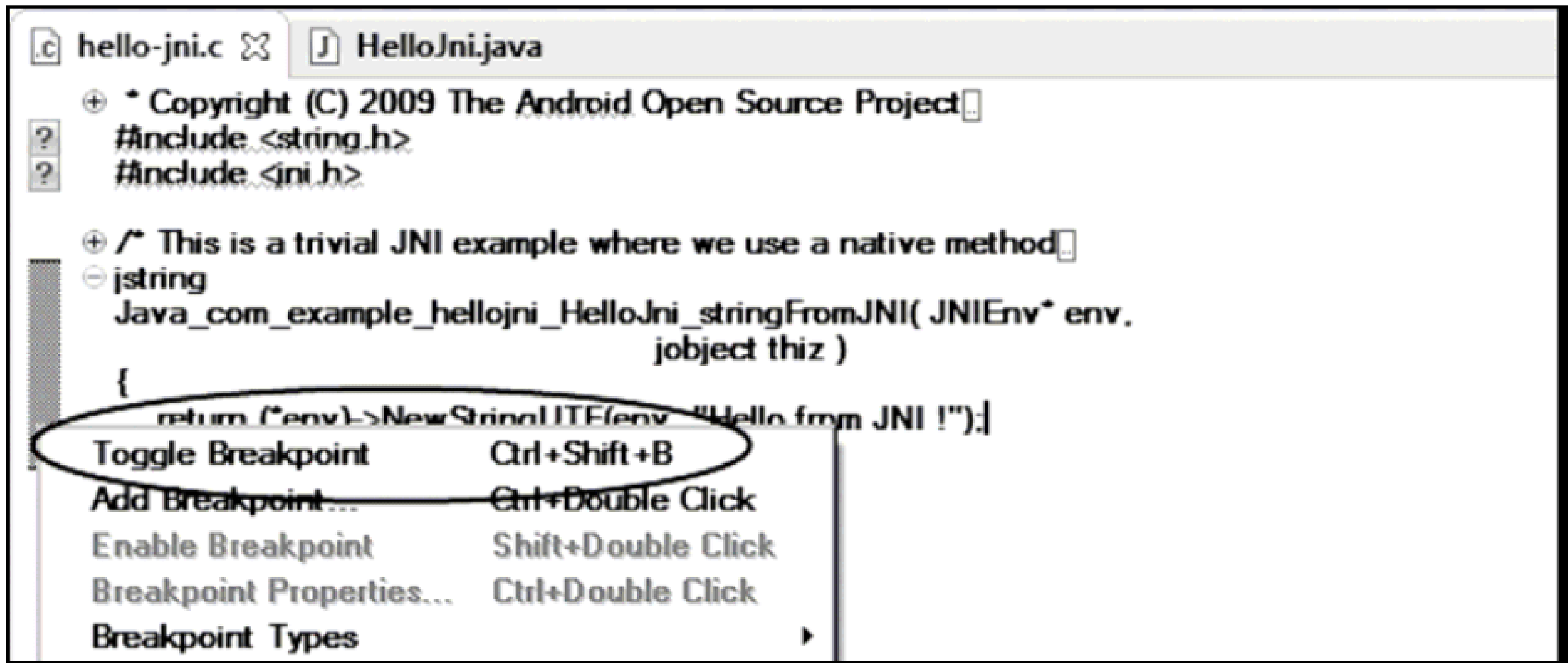


图 5-6　切换断点

小贴士：

用相同的上下文菜单，我们也可以放置一个条件断点以启用或禁用已经存在的断点。

(4) 既然断点已经设置成功，在主菜单上选择 Run | Debug Configurations 打开 Debug Configuration 对话框。

(5) 选择之前定义的调试配置。

(6) 单击 Debug 按钮。

(7) 针对不同的任务，Eclipse 支持不同的视图和工作台布局。在单击 Debug 按钮时，Eclipse 将询问用户是否切换调试视图，如图 5-7 所示。单击 Yes 继续执行。

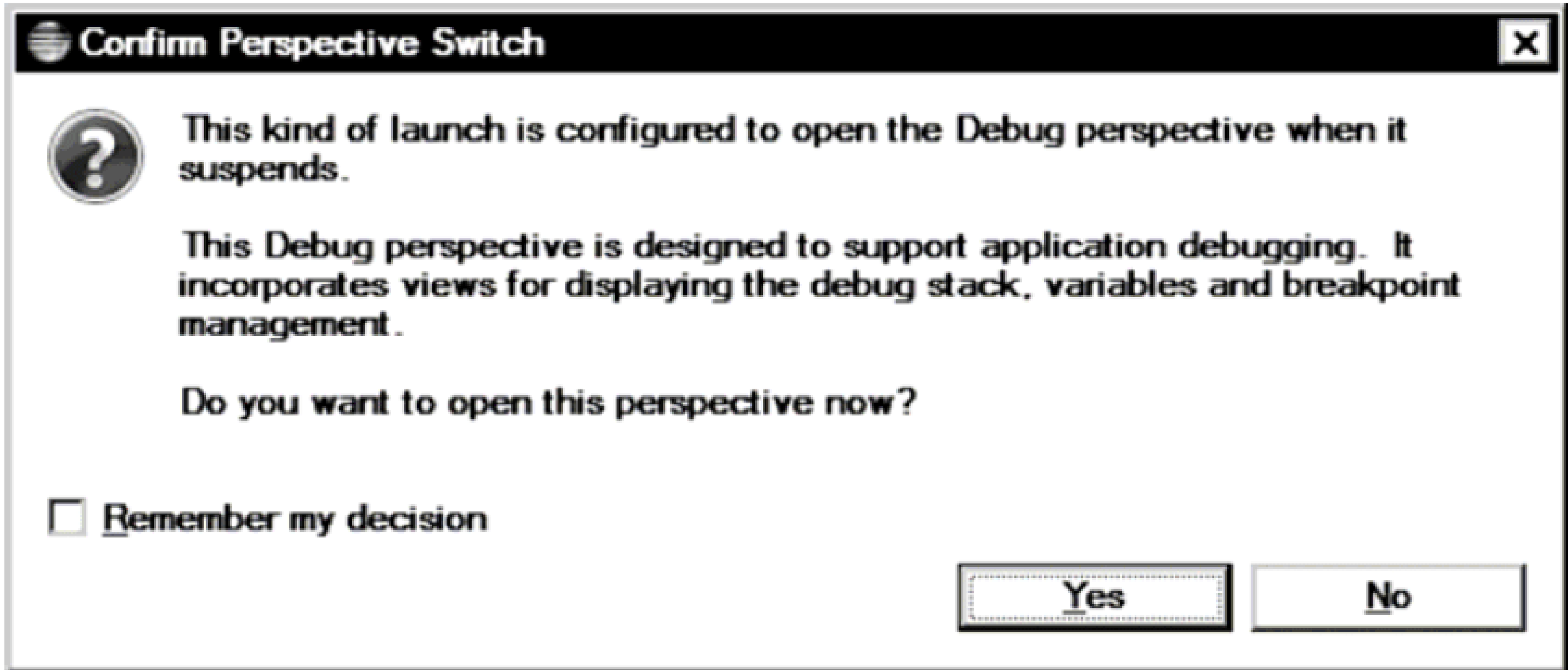


图 5-7　切换到调试情景

(8) 用 Android 设备或模拟器，单击 Call Native 按钮来调用原生函数。

原生代码一旦遇见断点，应用程序将停止并将控制权交给调试器，如图 5-8 所示。

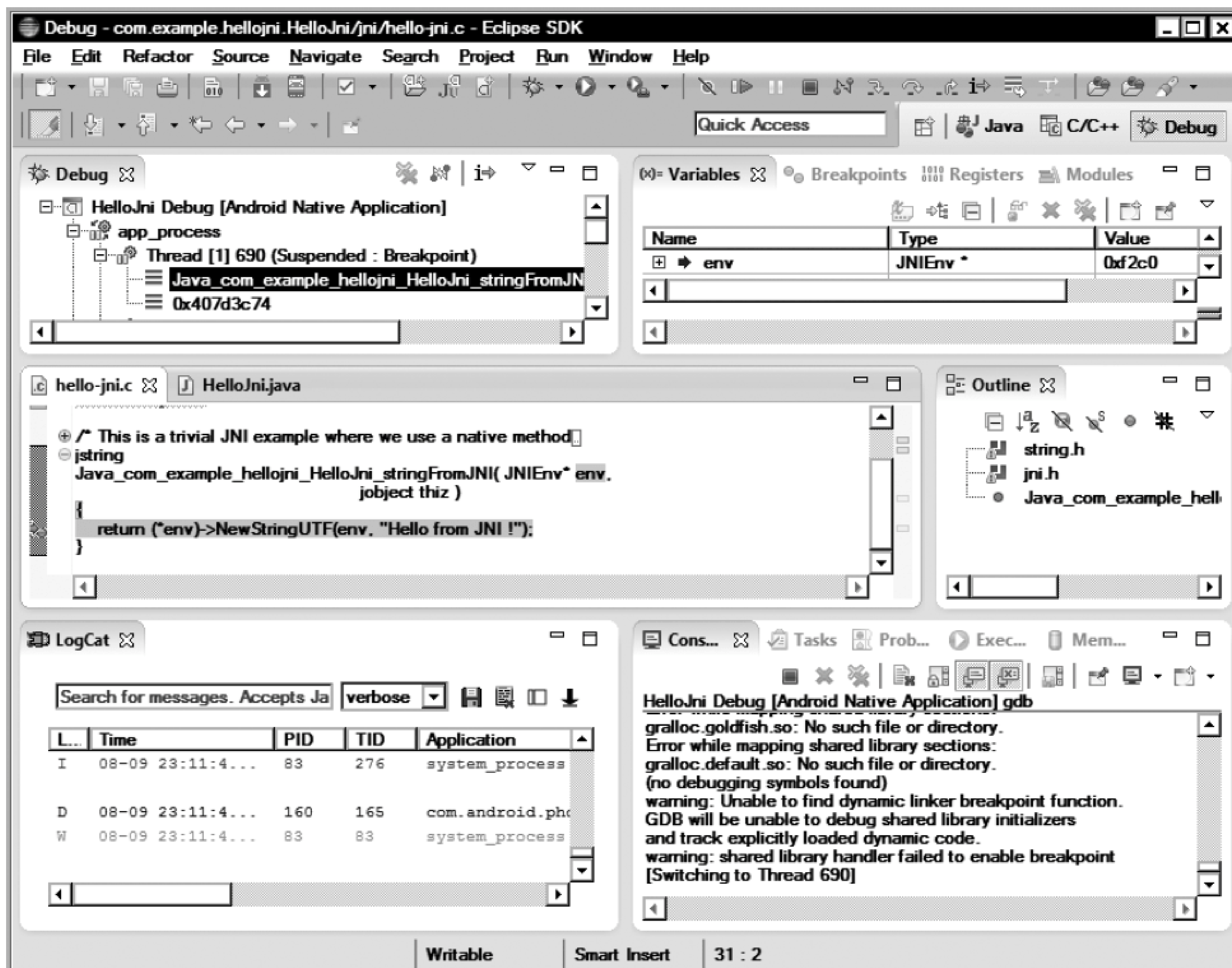


图 5-8　正在运行的 Eclipse 调试情景

调试情景将会显示一个原生代码当前状态的完整快照。在左上角，Debug 窗口显示正在运行的线程列表以及当前正在运行的函数。在右上角，Variables 视图让你访问原生变量并查看变量的当前值。中间区域的 Editor 视图中显示原生源代码，在下一步将要被执行的那行代码左侧的标记条上有一个箭头。可以用如图 5-9 所示的调试工具栏来控制应用程序的执行。



图 5-9　调试工具条

下面的操作是通过调试工具栏完成的：

- **Skip All Breakpoints**：使所有的断点无效。
- **Resume**：重新开始执行原生代码直到下一个断点。

- **Suspend：**通过向进程发送 SIGINT 中断信号来挂起正在执行的原生代码，此时可以检查原生代码的当前状态。
- **Step Into：**通过进入来执行下一个原生调用。
- **Step Over：**执行下一个原生调用，然后停止。
- **Step Return：**执行直至原生函数返回。
- **Terminate：**结束这个调试会话。

不仅可以用 Eclipse 调试原生应用程序，也可以用命令行方式实现相同级别的调试功能。

3.命令行

在命令行方式下，可以用 ndk-gdb 脚本调试原生代码，当前的 ndk-gdb 脚本要求运行 UNIX shell。在 Windows 平台上，可以用 Cygwin 代替命令提示符来调试。首先，打开基于所使用平台的 Cygwin 或终端窗口。可以使用 hello-jni 示例项目做这个实验。

(1) 为了防止发生冲突，确保 Eclipse 没有再运行。

(2) 将 hello-jni 项目目录设置成当前工作目录。

(3) 调用 rm-rf bin obj libs 命令删除 Eclipse 中的所有残留文件。

(4) 在命令行方式下调用 ndk-build 命令编译原生模块。

(5) 为了在命令行方式下编辑和打包应用程序，确保 ANT 版本的构建脚本文件 build.xml 存在于项目目录中。如果这是你第一次在命令行方式构建该项目，调用 android update project -p 命令生成必要的构建文件。如果使用 Cygwin，那么请用 android.bat 代替 android。

(6) 在命令行方式下调用 ant debug 命令在调试模式下编译和打包项目。

(7) 在命令行方式下调用 ant installd 命令将应用程序部署到设备或模拟器上。

(8) 默认情况下，ndk-gdb 脚本搜寻一个已经运行的应用程序进程；然而，可以用--start 或者--launch=<activity>参数在调试会话前自动启动应用程序。在命令行方式下调用 ndk-gdb --start 命令启动调试会话。当 GDB 成功依附于 hello-jni 应用程序时，显示 GDB 提示符。

(9) 在 GDB 提示符下，调用 b hello-jni.c:30 命令在 hello-jni.c 源文件的第 30 行增加一个断点。

(10) 既然断点已经定义，在 GDB 提示符下调用 c 继续执行原生应用程序。

(11) 在 Android 设备或模拟器中，单击 Native Call 按钮来调用原生函数。

注意：

看见一长串显示 GDB 不能定位多种系统库文件的错误消息是很正常的，你可以忽略那些消息，因为这些库文件的 symbol/debug 版本不可用。

当原生函数遇见断点时，应用程序将停止，此时可以用 GDB 检查原生代码的当前状态，如图 5-10 所示。

```
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
warning: shared library handler failed to enable breakpoint
(gdb) b hello-jni.c:30
Breakpoint 1 at 0x474d0c38: file jni/hello-jni.c, line 30.
(gdb) c
Continuing.
[New Thread 787]
[Switching to Thread 787]

Breakpoint 1, Java_com_example_hellojni_HelloJni_stringFromJNI (env=0xf2c0,
    thiz=0x413476b8) at jni/hello-jni.c:30
30          return (*env)->NewStringUTF(env, "Hello from JNI !");
(gdb) |
```

图 5-10　命令行调试会话

4. 有用的 GDB 命令

下面是可以在 GDB 提示下使用的、用来调试原生代码的有用 GDB 命令：

- **break<where>**：在指定的位置放置断点。位置可以是一个函数名、文件名或行数，如 file.c:10。
- **enable/disable/delete<#>**：激活、禁用或删除指定编号的断点。
- **clear**：清除所有断点。
- **next**：跳到下一个指令。
- **continue**：继续执行原生代码。
- **backtrace**：显示所有的堆栈。
- **backtrace full**：显示在每一个框架中包含局部变量的调用堆栈。
- **print<what>**：打印变量、表达式、内存地址或寄存器的内容。
- **display<what>**：和 print 相同，但在每一个指令步之后输出值。
- **what is <variable>**：显示变量的类型。
- **info threads**：列出所有运行中的线程。
- **thread <thread>**：在选择的线程上操作。
- **help**：返回所有命令列表的帮助信息。
- **quit**：结束调试会话。

注释：
当退出 GDB 提示符状态时，调试的应用程序将停止，这是一个众所周知的限制。

想了解更多关于 GDB 的信息请访问 www.gnu.org/software/gdb/documentation/查看 GDB 文档。

5.3　故障处理

在开发阶段，日志让你确定和展示应用程序的状态信息，这些信息将对之后解决问题有帮助。当通过日志记录的信息不够时但知道问题可能处在哪个环节时，调试就会发挥作

用。当你遇到意料之外的问题时，故障诊断技术将变成一个救星。了解正确的工具和技术将使你能够快速解决问题，本节简要介绍这部分内容。

5.3.1　堆栈跟踪分析

为了进行堆栈跟踪分析，需要在 hello-jni 示例应用程序中植入一个会引起故障的 bug。用 Eclipse 打开 hello-jni.c 源文件，按照程序清单 5-16 所示修改原生函数的内容。

程序清单 5-16　将 Bug 引入到原生函数

```
static jstring func1( JNIEnv* env )
{
    /* BUG 开始 */
    env = 0;
    /* BUG 结束 */

    return (*env)->NewStringUTF(env, "Hello from JNI !");
}
jstring
Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env,
                                                  jobject thiz )
{
    return func1(env);
}
```

通过将 JNIEnv 接口指针的值设置为 0，我们将触发这个故障。现在构建并运行该应用程序。当应用程序启动时，单击 Call Native 方法来调用原生函数。应用程序将中断，logcat 中将显示堆栈跟踪，如图 5-11 所示。

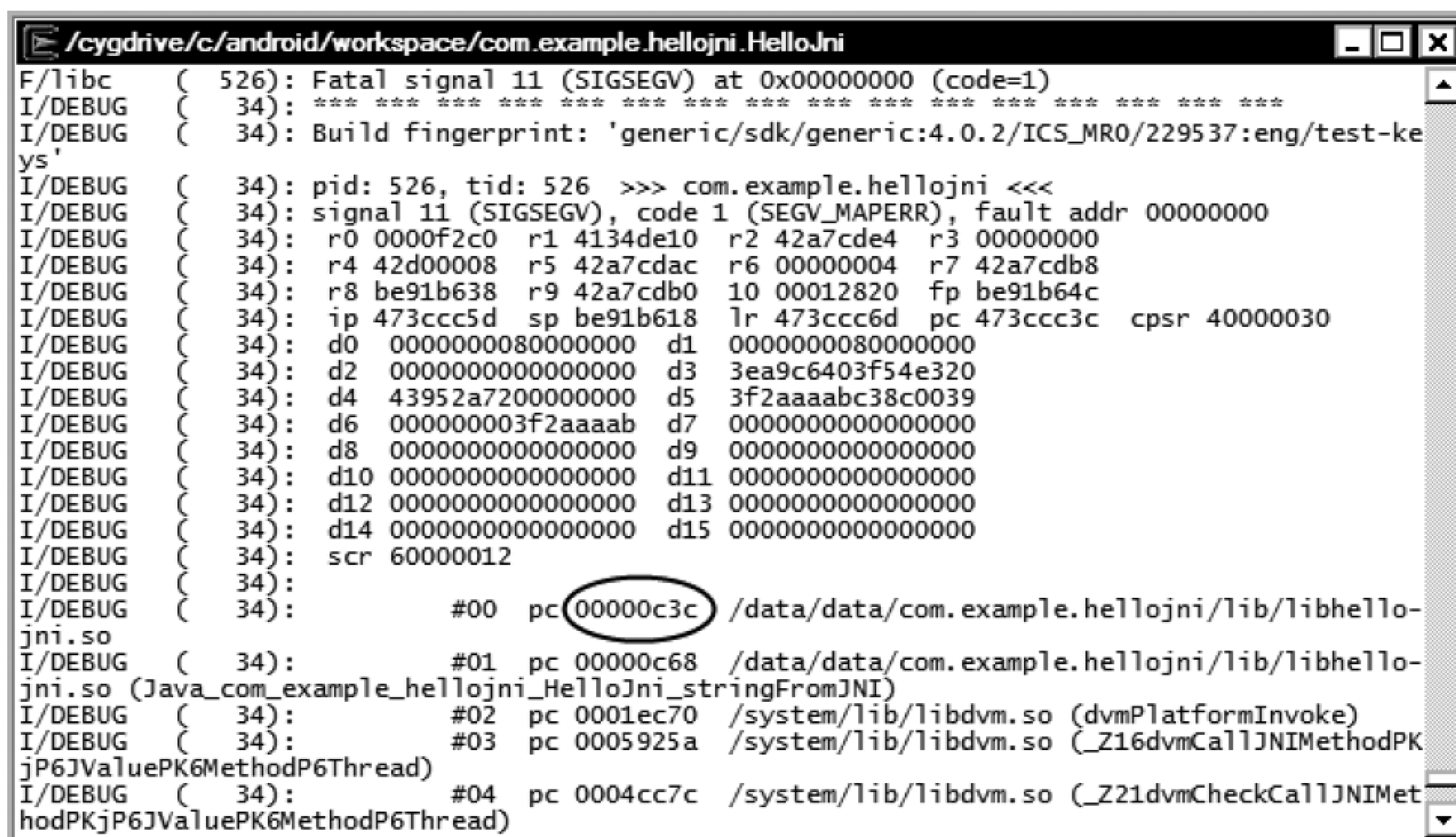


图 5-11　Logcat 显示故障后的堆栈跟踪

这些以井号(#)开头的行表示调用堆栈。第一行以#00 开头，是故障发生的地方；下一行#01 是以前的函数调用，以此类推。pc 后面跟的数字是代码的地址。如在堆栈跟踪中看到的，原生代码在地址为 00000c3c 处发生故障，前一个函数调用是 stringFromJNI 原生函数。地址 00000c3c 本身不能说明太多问题，但如果借助正确的工具，这个地址将被用来找到故障发生的确切文件和行号。Android NDK 带有一个叫做 ndk-stack 的工具，它能够把堆栈跟踪转换成确切的文件名和行号。在命令行方式下进入到项目根目录，执行以下命令：

```
adb logcat | ndk-stack =sym obj/local/armeabi
```

ndk-stack 工具将翻译堆栈跟踪，如图 5-12 所示。地址显示到了 jni/hello-jni.c 源文件的 33 行。有了这种信息，故障诊断容易多了。通过简单地在该地址放一个断点，可以停止应用程序并检查应用程序的状态。

```
/cygdrive/c/android/workspace/com.example.hellojni.HelloJni
$ adb logcat | ndk-stack -sym obj/local/armeabi
********** Crash dump: **********
Build fingerprint: 'generic/sdk/generic:4.0.2/ICS_MR0/229537:eng/test-keys'
pid: 526, tid: 526  >>> com.example.hellojni <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 00000000
Stack frame #00  pc 00000c3c  /data/data/com.example.hellojni/lib/libhello-jni.s
o: Routine func1 in jni/hello-jni.c:33
Stack frame #01  pc 00000c68  /data/data/com.example.hellojni/lib/libhello-jni.s
o (Java_com_example_hellojni_HelloJni_stringFromJNI): Routine Java_com_example_h
ellojni_HelloJni_stringFromJNI in jni/hello-jni.c:40
Stack frame #02  pc 0001ec70  /system/lib/libdvm.so (dvmPlatformInvoke)
Stack frame #03  pc 0005925a  /system/lib/libdvm.so (_Z16dvmCallJNIMethodPKjP6JV
aluePK6MethodP6Thread)
Stack frame #04  pc 0004cc7c  /system/lib/libdvm.so (_Z21dvmCheckCallJNIMethodPK
jP6JValuePK6MethodP6Thread)
Stack frame #05  pc 0005af84  /system/lib/libdvm.so (_Z22dvmResolveNativeMethodP
KjP6JValuePK6MethodP6Thread)
Stack frame #06  pc 00030a8c  /system/lib/libdvm.so
Stack frame #07  pc 000342ac  /system/lib/libdvm.so (_Z12dvmInterpretP6ThreadPK6
MethodP6JValue)
Stack frame #08  pc 0006c93e  /system/lib/libdvm.so (_Z15dvmInvokeMethodP6Object
PK6MethodP11ArrayObjectS5_P11ClassObjectb)
Stack frame #09  pc 00073d4a  /system/lib/libdvm.so
Stack frame #10  pc 00030a8c  /system/lib/libdvm.so
```

图 5-12　Ndk-stack 翻译代码地址

5.3.2　对 JNI 的扩展检查

默认情况下，JNI 函数基本不做错误检查。错误通常会导致故障。Android 为 JNI 调用提供了一个拓展的检查方式，被称之为 CheckJNI。当激活该功能时，JavaVM 和 JNIEnv 接口指针切换到函数表，这些函数表在调用实际的实现之前执行扩展错误检查。CheckJNI 能检测出以下问题：

- 企图分配负数大小的数组；
- 将错误的指针或 Null 指针传递给 JNI 函数；
- 传递类名称时语法错误；
- 在临界区调用 JNI；
- 给 NewDirectByeBuffer 传递错误参数；
- 当一个异常挂起时调用 JNI；

- 用在错误的线程中的 JNIEnv 接口指针；
- 域类型与 Set<Type>Field 函数不匹配；
- 方法类型与 Call<Type>Method 函数不匹配；
- 用错误的引用类型调用 DeleteGlobalRef/DeleteLocalRef；
- 错误的释放模式传递给 Release<Type>ArrayElement 函数；
- 从原生方法返回不兼容类型；
- 无效的 UTF-8 数列传递给 JNI 调用；

默认情况下，CheckJNI 模式只能在模拟器中使用，不能在常规的 Android 设备中使用，因为它会影响整个系统的性能。

启用 CheckJNI

在常规设备上，在命令行方式下执行以下命令可以启用 CheckJNI 模式：

```
adb shell setprop debug.checkjni 1
```

这不会影响运行中的应用程序，但是任何后来打开的应用程序都将启用 CheckJNI。CheckJNI 状态也显示在 logcat 中，如图 5-13 所示。

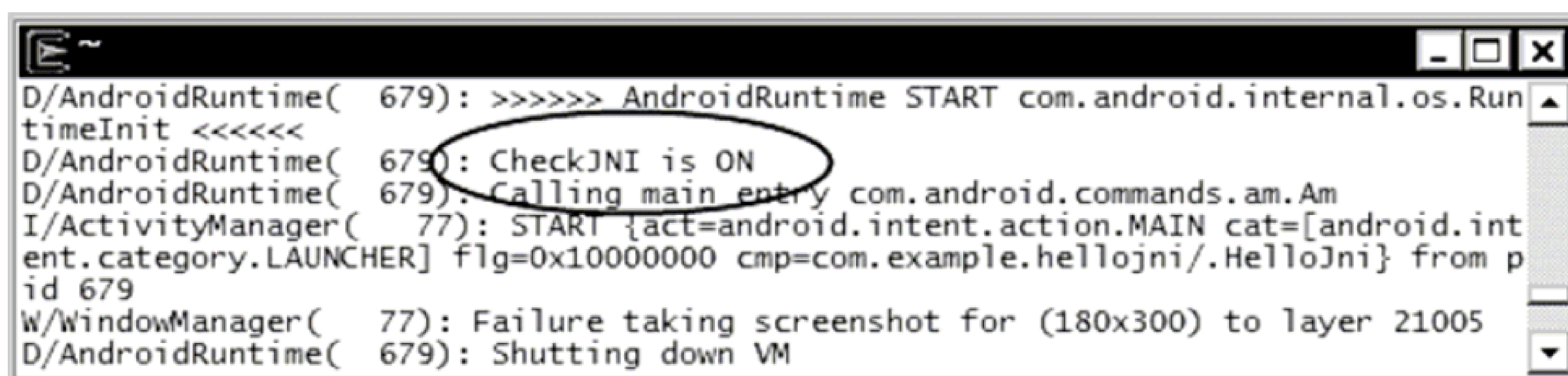


图 5-13　在 logcat 中显示 CheckJNI 的状态

为了实现 CheckJNI，用 Eclipse 打开 hello-jni.c 源代码。如程序清单 5-17 所示修改原生函数。

程序清单 5-17　根据原生大小创建数组

```
jstring
Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env,
                                                  jobject thiz )
{
    jintArray javaArray = (*env)->NewIntArray(env, -1);

    return (*env)->NewStringUTF(env, "Hello from JNI !");
}
```

你将用一个负数作为数组大小来创建一个新的实数数组。在模拟器上构建并运行应用程序。当应用程序启动时，单击 Call Native 按钮调用原生函数。如图 5-14 所示，CheckJNI 将在 logcat 中显示警告消息并且终止执行。

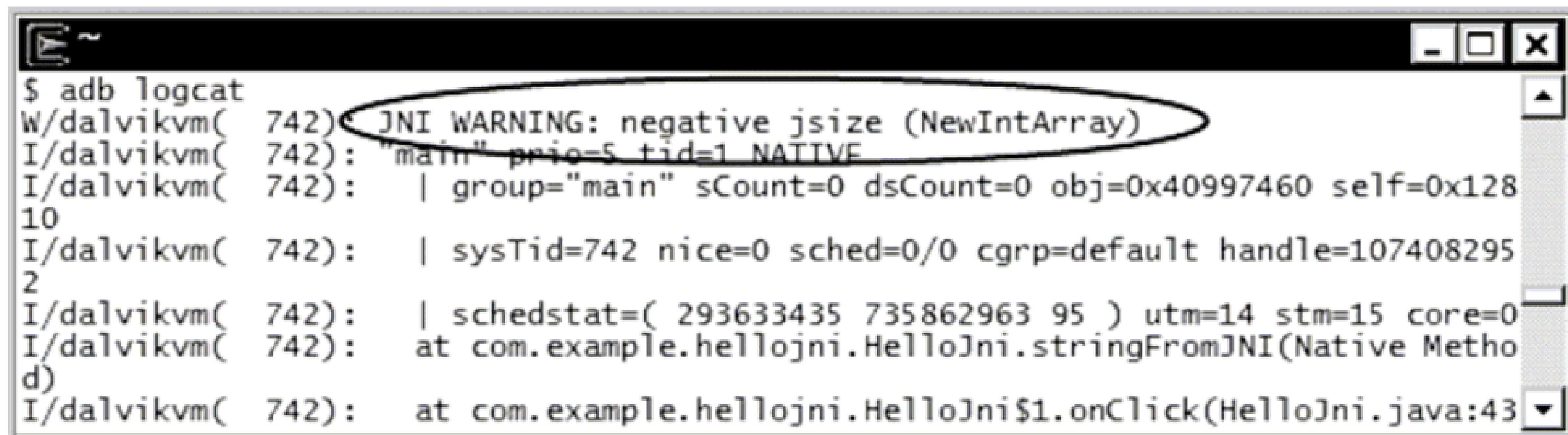


图 5-14　关于数组大小为负数的 JNI 警告

5.3.3　内存问题

在没有正确的工具时很难诊断出内存问题，本节将简要介绍两种诊断内存问题的方法。

1. 使用 libc 调试模式

如果使用模拟器，启用 libc 调试模式就能够诊断出内存问题。为了启用 libc 调试模式，使用程序清单 5-18 所示的命令。

程序清单 5-18　打开 libc 调试模式

```
adb shell setprop libc.debug.malloc 1
adb shell stop
adb shell start
```

支持 libc 调试模式的值是：

- **1**：执行泄露探测。
- **5**：填补分配的内存检查超支。
- **10**：填补内存和定点检查超支。

为了使用 Libc 调试模式，用 Eclipse 打开 hello-jni.c 源代码。如程序清单 5-19 所示修改原生函数。

程序清单 5-19　修改超出分配的缓存的内存

```
jstring
Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env,
                                                  jobject thiz )
{
    char* buffer;
    size_t i;

    buffer = (char*) malloc(1024);
    for (i = 0; i < 1025; i++)
    {
       buffer[i] = 'a';
```

```
        }

        free(buffer);

        return (*env)->NewStringUTF(env, "Hello from JNI !");
    }
```

你将分配到 1 024 字节，但是代码将修改所分配内存之外的额外内存，这会导致内存崩溃。执行程序清单 5-20 所示的命令打开 libc 调试模式。

程序清单 5-20　为内存崩溃检测开启 libc 调试模式

```
adb shell setprop libc.debug.malloc 10
adb shell stop
adb shell start
```

在模拟器上构建并运行应用程序。当应用程序启动的时候，单击 Call Native 按钮调用原生函数。如图 5-15 所示，libc 调试模式将在 logcat 中显示一个关于内存崩溃的警告消息并终止执行。

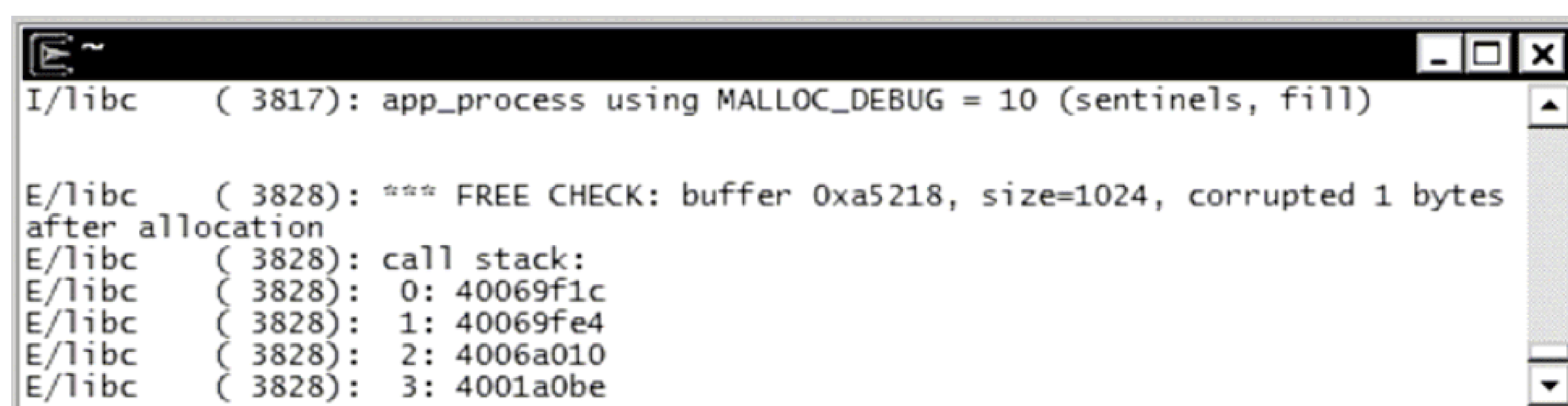


图 5-15　Libc 调试模式下显示内存崩溃错误

2. Valgrind

Libc 调试模式可以对内存问题做基本的故障诊断，Valgrind 可以用来进行更高级的内存分析。它是一个用于内存调试、内存泄漏检测和概要分析的开源工具。要完成这个实验，可以在本书提供的网址下载 prebuilt Valgrind 二进制文件或在你自己的机器上构建。如果你想构建，跳到“从源代码构建”一节。

(1) 使用 Prebuilt 二进制文件

用你的 web 浏览器，从 http://zdo.com/valgrind-arm-emulator-3.8.0.zip 下载 ARM 模拟器下的 Valgrind 二进制文件压缩文件。解压缩 ZIP 文件的内容并把它的位置记录下来，跳到“安装到模拟器”一节。

(2) 从源代码构建

为了从源代码中为 Android 系统建立合适的 Valgrind，需要一个 Linux 宿主系统。Valgrind 的官方发行版本支持 Android。从 http://valgrind.org/downloads/current.html 下载最新版本的 Valgrind。在本书编写时，Valgrind 的最新版本是 3.8.0，它是一个 BZIP2 压缩的 TAR 文件。在命令行方式下，执行以下命令解压缩。

```
tar jxvf valgrind-3.8.0.tar.bz2
```

在解压缩 Valgrind 源文件时，使用你的编辑器打开 README.android 文件查看最新的构建指令。因为在 Android 模拟器中使用 Valgrind，请确认通过执行下列命令将 HWKIND 的值设置成 emulator。

```
export HWKIND=emulator
```

在建立合适的 Valgrind 时，二进制文件和其他必备组件将被放置在 Inst 子目录中。

(3) 将 Valgrind 部署到模拟器

在使用 Valgrind 之前需要将其部署到模拟器中。为了完成此项任务，打开 Cygwin 或一个终端窗口。如果你使用的是预构建二进制文件，请进入解压缩文件的根目录；如果从源代码构建，进入 Valgrind 源代码根目录，在命令行方式下执行下列命令：

```
adb push Inst /
```

这将把 Valgrind 文件部署到模拟器上的/data/local/Inst 目录下。在将文件部署到设备上时，需要修改执行位。为此执行以下命令：

```
adb shell chmod 755 \
    $(find Inst -type f -exec file {} \; | \
                    grep executable | \
                    sed -n -e 's/^Inst\([^:]*\).*$/\1/gp' | \
                    xargs)
```

(4) Valgrind 包装器

除了 Valgrind 二进制文件，还需要一个帮助脚本。使用 Eclipse 或者你喜欢的编辑器，用程序清单 5-21 显示的内容创建名为 valgrind_wrapper.sh 的新文件。

程序清单 5-21　Valgrind 包装器 Shell 脚本

```
#!/system/bin/sh

export TMPDIR=/sdcard
exec /data/local/Inst/bin/valgrind --error-limit=no $*
```

修改包装器脚本的最后一行时，把它部署到模拟器上，并执行程序清单 5-22 所示的命令授予命令的可执行权限。

程序清单 5-22　部署 Valgrind 包装器脚本

```
dos2unix.exe valgrind_wrapper.sh
adb push valgrind_wrapper.sh /data/local/Inst/bin
adb shell chmod 755 /data/local/Inst/bin/valgrind_wrapper.sh
```

(5) 运行 Valgrind

为了在 Valgrind 下运行应用程序，执行程序清单 5-23 所示的命令在启动序列中引入包装器脚本。

程序清单 5-23　在启动序列中引入 Valgrind 包装器

```
adb shell setprop wrap.com.example.hellojni \
"logwrapper /data/local/Inst/bin/valgrind_wrapper.sh"
```

该属性键的格式是 wrap.<package name>。为了在 Valgrind 下运行其他应用程序，只要用合适的值替代包名即可。停止然后重启应用程序，Valgrind 消息将被展示在 Logcat 上，如图 5-16 所示。



图 5-16　Logcat 显示 Valgrind 消息

注意：

在 Valgrind 下运行应用程序将大幅度地降低应用程序的运行速度。Android 系统可能会抱怨进程没有响应。在这种情况下，请单击 Wait 按钮给 Valgrind 更多的时间。

5.3.4　strace

在某些情况下，你可能想在没和调试器相连也不增加大量日志消息的情况下监控应用程序的每个活动。用 strace 工具可以很容易的实现该功能。strace 工具是有用的诊断工具，因为它拦截并记录应用程序调用的系统调用以及收到的信号。每一个系统调用的名字、参数及返回值都会被输出。注意 Android 模拟器带有 strace 功能。

为了使用 strace，用 Eclipse 打开 hello-jni.c 源代码。按照程序清单 5-24 的内容修改源文件。

程序清单 5-24　增加了两个系统调用的原生源代码

```
#include <unistd.h>
...
jstring
Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env,
                                                  jobject thiz)
{
    getpid();
    getuid();

    return (*env)->NewStringUTF(env, "Hello from JNI !");
}
```

在模拟器上构建并运行应用程序。打开 Cygwin 或一个终端窗口，当应用程序启动时，执行下列命令以获得应用程序的进程 ID：

```
adb shell ps | grep com.example.hellojni
```

第三列的数字是进程 ID，如图 5-17 所示。

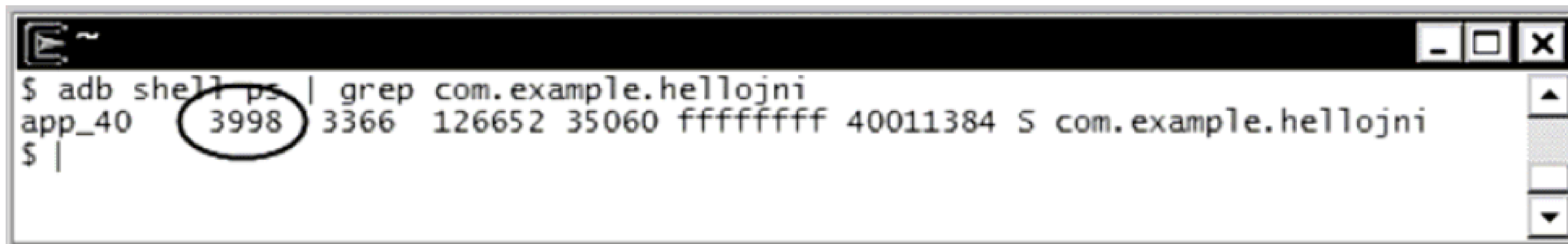


图 5-17　获得应用程序的进程 ID

执行以下命令，通过替代进程 ID 使 strace 联系到运行中的应用程序进程：

```
adb shell strace -v -p <Process ID>
```

如你所见，strace 将被链接到应用程序的进程，并且它将拦截并显示系统调用、参数及其返回值。单击 Call Native 按钮调用原生函数，strace 将显示两个引入到原生代码中的系统调用，如图 5-18 所示。

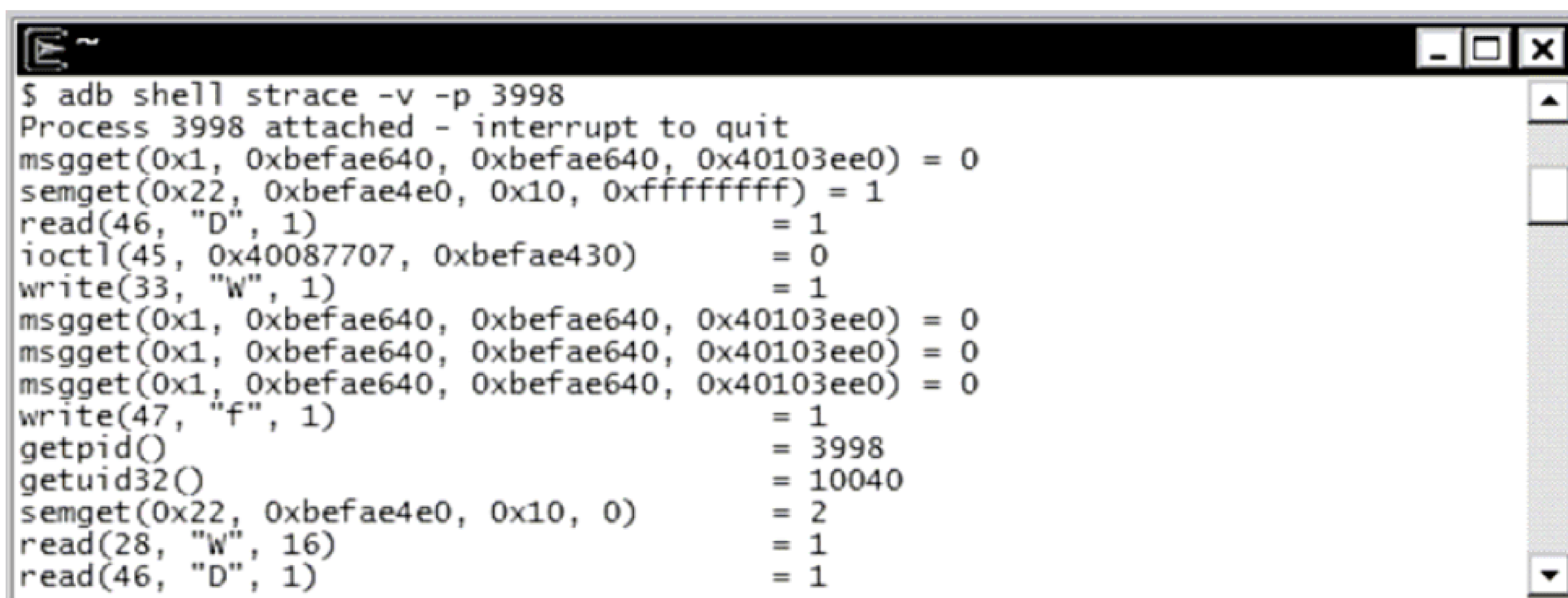


图 5-18　strace 显示系统调用

无论是打开还是关闭代码应用程序，strace 都是有效的故障诊断工具。

5.4　小结

本章学习了 Android 平台上的日志、调试和故障诊断的工具和技术。在以后的章节中你会发现，用 Android 平台提供的原生 APIs 进行实验时，本章给出的概念会非常有用。

第 6 章

Bionic API 入门

第 5 章学习了有关 Android 原生应用程序开发的日志、调试以及故障排除工具和技术。从本章开始，我们将深入探讨 Android NDK 提供的原生 API。

Bionic 是 Android 平台为使用 C 和 C++进行原生应用程序开发所提供的 POSIX 标准 C 库。它是 Google 为 Android 操作系统提供的 BSD 标准 C 库的衍生库。它由带有可处理线程、进程和信号的定制 Linux 专用位的混合 BSD C 库文件片组成，“Bionic”的名称就是来源于此。

Bionic 是原生应用程序开发的重要内容，因为它提供了在 Android 平台上开发任何类型的功能性原生代码所需要的最小构造集。在以后的章节中，我们将在很大程度上依赖于 Bionic 所提供的功能。在了解 Bionic 的具体内容之前，我们先快速回顾一下一般的标准库。

6.1 回顾标准库

编程语言的标准库提供了经常要用到的构造、算法、数据结构以及任务的抽象接口，这些任务通常主要涉及硬件和操作系统，例如网络访问、多线程、内存管理和文件 I/O 等。由于不同编程语言的特性差异很大，因此它们的标准库千差万别，可以是非常小的一组只适用于重要任务的构造，也可以非常广泛、功能强大。无论如何，为了给应用程序开发提供方便，每种编程语言的实现都提供标准库。

几乎每一种编程语言都有一个标准库。Java 平台的标准库是 Java Class Library(JCL)，它是一个 Java 编程语言的标准库，包含了一组适用于排序、字符串操作等常见操作的全面的标准类库和一个访问底层操作系统服务的抽象接口，例如可用于文件与网络交互的 I/O 流。Android 框架通过结合 Android 应用程序开发特有的附加构造扩展了 JCL。

对 C 语言来说，ANSI C 标准定义了标准库的范围。该标准库被称为 C 标准库，或简称为 libc。C 语言的实现也伴随着 C 标准库的实现。除了标准 C 库的规范，POSIX C 库的规范声明了附加结构，在 POSIX 兼容系统中，这些附加结构应该包含在此类标准库里。

6.2 还有另一个 C 库

Google 创建一个新的 C 库而不复用现有的 GNU C 库(glibc)或 Embedded Linux C 库(uClibc)的动机可总结为以下三点：

- 许可：glibc 和 uClibc 在 GNU Lesser General Public License (LGPL，GNU 宽通用公共许可)下均可用，从而限制了可被专用应用程序使用的方式。相反，Bionic 是在 BSD 许可下发布的，该许可非常自由，不对库的使用做任何限制。
- 速度：Bionic 是专门为移动计算精心设计的。针对移动设备上有限的 CPU 周期和可用内存进行了裁剪以提高工作效率。
- 大小：Bionic 的核心理念是简单化。它提供了针对内核功能的轻量级包装器和一组较小的 API，使得它比其他的代替品小。本章将会介绍这些 API。

6.2.1 二进制兼容性

尽管 Bionic 是 C 标准库，但它不以任何方式与其他 C 库二进制兼容。用其他 C 库生成的目标文件和静态库不应该与 Bionic 进行动态链接，这么做通常会导致无法链接或无法正确执行你的原生应用程序。

除此之外，任何与其他 C 库静态链接生成的、且不与 Bionic 混合的应用程序都无可争议地可以在 Android 平台上运行，除非它在运行时动态加载了其他系统库。

6.2.2 提供了什么

Bionic 提供 C 标准库宏、类型定义、函数和少数 Android 特有的特性，这些特性可分项在以下功能域中列出：

- 内存管理
- 文件输入输出
- 字符串操作
- 数学
- 日期和时间
- 进程控制
- 信号处理
- 网络套接字
- 多线程
- 用户和组
- 系统配置
- 命名服务切换

6.2.3　缺什么

如前所述，Bionic 是专门为 Android 平台及进行移动计算而设计的。Bionic 不会支持所有 C 标准库的函数。Android NDK 文档提供了缺失功能的完整列表；然而，可以在实际的头文件中获取这些信息。Bionic 头文件可以放到 Android_NDK_HOME 目录下的 platforms/android-<api-level>/arch-<architecture> /usr/include 中。

该目录下的每个头文件中都包含一个部分，该部分清楚地标明了缺失函数列表。如程序清单 6-1 所示，该部分列出了 stdio.h 头文件中缺失的函数。

程序清单 6-1　Bionic 实现的缺失函数

```
#if 0 /* 缺失自 BIONIC */
char *ctermid(char *);
char *cuserid(char *);
#endif /* 缺失 */
```

预处理器 if 语句被用来禁用头文件中的这些行，而相关的注释则表明该部分包含了缺失函数列表。除了这个列表，Android NDK 文档还列举了一些通过 Bionic 展示的函数，但这些函数只作为存根被实现，没有任何功能或只有很少功能。

6.3　内存管理

内存是进程最基本的可用资源。对于 Java 应用程序来说，由虚拟机来管理内存。在创建新对象时分配内存，未使用的内存会通过垃圾回收器自动归还给系统。然而在原生空间中，应用程序要显式地管理它们自己的内存。在原生应用程序开发过程中，有效地管理内存是非常重要的，因为做不好将导致系统可用内存耗尽，并且会在总体上严重影响应用程序和系统稳定性。

6.3.1　内存分配

C/C++程序语言支持三种内存分配方式：

- **静态分配**：适用于在代码中定义的每个静态和全局变量，静态分配在应用程序启动时自动发生。
- **自动分配**：适用于每个函数参数和局部变量，自动分配在包含声明的复合语句被输入时发生；退出复合语句时所分配的内存被自动释放。
- **动态分配**：静态分配和自动分配都假设需要的内存大小和范围是固定的，且在编译时就被定义。而动态分配则在事先不知情的情况下起作用，这些内存大小和范围的分配取决于运行时因素。

本节将介绍基于 C 和 C++应用程序的动态内存管理。

6.3.2 C 语言的动态内存管理

C 语言不提供对内置动态内存管理的支持。Bionic C 库则提供了一组函数以使 C 代码中可以使用动态内存。

1.在 C 中分配动态内存

C 语言中，可以在运行库用标准 C 库函数 malloc 分配动态内存。

```
void* malloc(size_t size);
```

为了用这个函数，首先应该包含 stdlib.h 标准 C 库头文件。如程序清单 6-2 所示，malloc 只有一个参数，分配的内存大小以字节为单位，并返回一个指向新分配内存的指针。

程序清单 6-2　C 代码中用 malloc 的进行动态内存分配

```
/* 包含标准 C 库头文件. */
#include < stdlib.h>

...

/* 分配 16 个元素的整型数组. */
int* dynamicIntArray = (int*) malloc(sizeof(int) * 16);
if (NULL == dynamicIntArray) {
    /*不能分配足够的内存. */
    ...
} else {
    /* 通过整型指针使用内存. */
    *dynamicIntArray = 0;
    dynamicIntArray[8] = 8;
    ...
    /* 释放分配的内存. */
    free(dynamicIntArray);
    dynamicIntArray = NULL;
}
```

提示：
因为 malloc 是以字节数为单位分配内存的，所以可以用 C 关键字 sizeof 来提取数据类型大小。

如果所请求的内存大小不能满足，malloc 会返回 NULL 以说明这种情况。应用程序在使用 malloc 之前应该先检查其返回值。一旦分配，普通 C 代码便可通过指针使用动态内存直到内存被释放。

2. 在 C 语言中释放动态内存

当不再需要动态内存时，应用程序应该显式地释放它。可以用标准 C 库函数 free 释放动态内存。

```
void free(void* memory);
```

free 函数用指向之前分配的动态内存的指针作参数并释放相应内存，如程序清单 6-3 所示。

程序清单 6-3　在 C 代码中用 free 释放动态内存

```
int* dynamicIntArray = (int*) malloc(sizeof(int) * 16);
...
/* 使用分配的内存. */
...
free(dynamicIntArray);
dynamicIntArray = NULL;
```

需要注意的是，在函数调用后即使该指针指向的内存已经被释放，但指针的值并不改变。任何试图使用该无效指针的动作都会引起分段违规。为了避免意外使用无效指针，最好在释放指针后立刻将它置为 NULL。

3.改变 C 语言中的动态内存分配

一旦分配内存，可以用标准 C 库提供的 realloc 函数来改变内存大小。

```
void* realloc(void* memory, size_t size);
```

动态分配的内存大小可以根据新的大小被扩展或缩小。realloc 函数将分配的原始动态内存作为第一个参数，新的内存大小作为第二个参数，如程序清单 6-4 所示。

程序清单 6-4　用 realloc 重新分配已分配的动态内存

```
int* newDynamicIntArray = (int*) realloc(
        dynamicIntArray, sizeof(int) * 32);

if (NULL == newDynamicIntArray) {
    /* 不能重新分配足够的内存. */
    ...
} else {
    /* 更新内存指针. */
    dynamicIntArray = newDynamicIntArray;
    ...
}
```

realloc 函数返回指向重新分配的内存的指针。该函数可以通过保存内容将原来的内存移到一个新的位置上，此时将返回新位置。如果函数调用失败，将保留原来的动态内存分配不变并返回 NULL。

6.3.3　C++的动态内存管理

C++提供了对动态内存管理的内置支持，可以采用 C++的 new 和 delete 关键字而不是标准 C 库函数来管理动态内存分配。

当处理 C++对象时，强烈建议用 C++关键字而不是标准 C 库提供的函数。和标准 C 库函数不同，C++动态内存管理关键字是类型敏感的，且它们支持 C++对象生命周期。除了分配内存，new 关键字也调用类的构造函数；同样的，delete 关键字在释放内存之前先调用类的析构函数。

1. C++中动态内存的分配

用 new 关键字后面紧跟着数据类型来分配内存，如程序清单 6-5 所示。

程序清单 6-5　C++代码中单个元素的动态内存分配

```
int* dynamicInt = new int;
if (NULL == dynamicInt) {
    /* 不能分配足够的内存. */
    ...
} else {
    /* 使用已分配的内存. */
    *dynamicInt = 0;
    ...
}
```

如果要分配数组元素，数组元素的个数要用方括号指定，如程序清单 6-6 所示。

程序清单 6-6　C++代码中的多个元素的动态内存分配

```
int* dynamicIntArray = new int[16];
if (NULL == dynamicIntArray) {
    /* 不能分配足够的内存. */
    ...
} else {
    /* 使用已分配的内存. */
    dynamicIntArray[8] = 8;
    ...
}
```

2. 释放 C++中的动态内存

当不再需要动态内存时，应该由应用程序使用 C++的 delete 关键字显式释放，如程序清单 6-7 所示。

程序清单 6-7　使用 delete 关键字释放单个元素的动态内存

```
delete dynamicInt;
dynamicInt = 0;
```

如果要释放数组元素，应该用 C++的关键字 delete[]，如程序清单 6-8 所示。

程序清单 6-8　使用 delete[]释放数组动态内存

```
delete[] dynamicIntArray;
dynamicIntArray = 0;
```

注意，要正确使用 delete 关键字；否则会导致原生应用程序内存泄漏。

3. 改变 C++的动态内存分配

C++中不提供重新分配动态内存的内置支持。内存分配是根据数据类型大小和元素个数来定的。如果应用程序逻辑上需要在运行时增加或减少元素个数，强烈建议使用适当的标准模板库(Standard Template Library，STL)容器类。

4. 混合内存函数和关键字

在处理动态内存时，开发人员必须使用正确的函数和关键字对。通过 malloc 分配的内存块必须用 free 关键字释放；同样的，通过 new 关键字分配的内存块必须相应地由 delete 关键字释放。否则会导致未知的应用程序行为。

6.4　标准文件 I/O

原生应用程序可以通过标准 C 库提供的 Standard File I/O (stdio)函数与文件系统交互。标准 C 库提供了两种文件 I/O：

- **低级 I/O**：原始的 I/O 函数，有更完善的数据源控制等级。
- **流 I/O**：更高级别的、可缓冲的 I/O 函数，更适合处理数据流。

在处理常规文件时，基于 I/O 的流会更加灵活和方便。本节将重点讲解流 I/O 函数，在本章的 socket 通信部分将涵盖部分低级 I/O 函数。

6.4.1　标准流

有三种预定义的流 I/O 可以在原生代码通过流 I/O 立即使用，这些流分别表示原生应用程序的标准输入输出通道，在标准 I/O 头文件中它们被定义为下列变量。

(1) **stdin**：应用程序标准输入流。

(2) **stdout**：应用程序标准输出流。

(3) **stderr**：应用程序标准错误流。

由于 Android 上的原生应用程序是图形用户界面(graphical user interface，GUI)下运行的一个模块，这些流并不是很有用。当与旧代码集成时，必须确保任何对这些标准流的使用都通过 GUI 妥善处理。正如第 5 章中“控制台日志”一节所讲的，在启动应用程序前设置 log.redirect-stdio 系统属性可以将 stdout 和 stderr 流导向 Android 系统日志。

6.4.2　使用流 I/O

流 I/O 的构造和函数在 stdio.h 标准 C 库头文件已经定义过。为了在原生应用程序中使用流 I/O，事先应该包含这个头文件，如程序清单 6-9 所示。

程序清单 6-9　包含标准 I/O 头文件用以使用 I/O 流

```
#include <stdio.h>
```

由于一些历史原因，在标准 C 库中，表示流的数据结构类型被称为 FILE，不是流。一个 FILE 对象保存了流 I/O 连接的所有内部状态信息，流 I/O 函数创建并维持 FILE 对象，且应用程序代码不会直接处理该对象。

6.4.3 打开流

可以通过流 I/O 的 fopen 函数打开一个新文件或现有文件的新流。fopen 函数将文件名称和打开类型作为参数，并返回一个流指针。

```
FILE* fopen(const char* filename, const char* opentype);
```

fopen 函数的第二个参数 opentype 是一个控制文件打开方式的字符串。它应该由以下打开类型的其中一个开头：

- **r**：以只读方式打开现有文件。
- **w**：以只写方式打开现有文件。如果该文件已经存在，它会被截断，截断后文件长度为 0。
- **a**：以附加方式打开文件。保存文件内容，新输出的内容附加到文件结尾处。如果该文件不存在，将会打开一个新的文件。
- **r+**：在读写模式下打开文件。
- **w+**：在读写模式下打开文件。如果该文件已经存在，它会被截断，截断后文件长度为 0。
- **a+**：打开文件进行读取和附加。在读取时，初始文件的位置被设定在开头，而附加时被设定在文件结尾。

注意：

若文件是以 r+、w+或 a+双模式打开的，在读写转换之前应先用 fflush 函数刷新缓冲区。

若文件不能以请求的模式打开，fopen 函数会返回一个 NULL 指针。如果成功，一个流指针(一个 FILE 指针)会被返回以便与流交互，如程序清单 6-10 所示。

程序清单 6-10　以只写模式打开一个流

```
#include <stdio.h>
...
FILE* stream = fopen("/data/data/com.example.hellojni/test.txt", "w");
if (NULL == stream)
{
    /* 写文件打不开. */
}
else
{
    /*使用流. */

    /* 关闭流. */
}
```

流被打开后，就可以用来读和写直到它被关闭。

6.4.4　写入流

流 I/O 为流的写入提供了 4 个函数，本节将简要介绍这些函数。

1. 向流中写入数据块

可以用 fwrite 函数向流中写入数据块。

```
size_t fwrite(const void* data, size_t size, size_t count, FILE* stream);
```

如程序清单 6-11 所示，fwrite 函数从缓冲区 data 向给定的流 stream 中写 count 个大小为 size 的元素。

程序清单 6-11　用 fwrite 向流中写数据块

```
char data[] = { 'h', 'e', 'l', 'l', 'o', '\n' };
size_t count = sizeof(data) / sizeof(data[0]);

/* 向流中写数据. */
if (count ! = fwrite(data, sizeof(char), count, stream))
{
    /* 向流中写数据时产生错误. */
}
```

它会返回实际写入流的元素个数。如果成功，返回的值应该等于 count 所指定的个数；否则表示写时有错误。

2. 向流中写入字符序列

可以用 fputs 函数向流中写入以 null 结尾的字符序列。

```
int fputs(const char* data, FILE* stream);
```

如程序清单 6-12 所示，fputs 函数将给定的字符序列数据写入名为 stream 的给定的流。

程序清单 6-12　用 fputs 向流中写字符序列

```
/* 向流中写字符序列. */
if (EOF == fputs("hello\n", stream))
{
    /*向流中写数据时产生错误. */
}
```

如果字符序列不能被写入流，fputs 函数返回 EOF。

3. 向流中写入单个字符

可以用 fputc 函数向流中写入一个单个字符或字节。

```
int fputc(int c, FILE* stream);
```

如程序清单 6-13 所示，fputc 函数在写入给定的流之前，将单个字符 c 作为一个整数并将其转换为一个无符号字符，该流命名为 stream。

程序清单 6-13　用 fputc 向流中写一个单个字符

```
char c = 'c';

/* 向流中写一个单个字符. */
if (c ! = fputc(c, stream))
{
    /* 向字符串中写字符时产生错误.*/
}
```

如果字符不能被写入流，则 fputc 函数返回 EOF；否则会返回字符本身。

4. 向流中写入有格式的数据

可以用 fprintf 函数在给定的流中格式化并输出可变数量的参数。

```
int fprintf(FILE* stream, const char* format, ...);
```

在所引用的格式中包括一个指向流的指针、格式字符串和可变数量的参数。格式字符串由普通字符和格式说明符混合而成。其中的普通字符被原样传送到流中，格式说明符使 fprintf 函数格式化并相应地将给出的参数写入流中。使用最频繁的说明符有：

- **%d、%i**：将整数参数格式化为有符号十进制数
- **%u**：将无符号整数格式化为无符号十进制数
- **%o**：将无符号整数参数格式化为八进制
- **%x**：将无符号整数参数格式化为十六进制
- **%c**：将整数参数格式化为单个字符
- **%f**：将双精度参数格式化为浮点数
- **%e**：将双精度参数格式化为固定格式
- **%s**：打印给出的 NULL 结尾字符数组
- **%p**：打印给出的指针作为内存地址
- **%%**：写入一个%字符

如程序清单 6-14 所示，fprintf 函数提供的参数顺序和类型应该与格式字符串中的说明符相匹配。

程序清单 6-14　向流中写带格式的数据

```
/* 写带格式的数据. */
if (0 > fprintf(stream, "The %s is %d.", "number", 2))
```

```
{
    /* 写带格式的数据时产生错误. */
}
```

fprintf 函数返回写入流中的字符个数。在出错的情况下，它返回一个负数。关于格式字符串的更多信息，包括字符串格式说明符和修改符的完整列表，请参见 http://pubs.opengroup.org/onlinepubs/009695399/functions/fprintf.html 中的 fprintf 使用说明页。

5. 刷新缓冲区

流 I/O 积累写入的数据并异步地将其传送至底层文件中，而不是立即将数据写入文件。类似的，流 I/O 从文件中以块的方式读取数据而不是逐个字符地读，这就是所谓的缓冲。

刷新缓冲区意味着将所有积累的数据传送到底层文件中。在以下情况下刷新会自动进行：

- 应用程序正常终止
- 在行缓冲时写入新行
- 当缓冲区已满
- 当流被关闭

流 I/O 也提供了 fflush 函数使得应用程序可以在需要时手动刷新缓冲区。

```
int fflush(FILE* stream);
```

如程序清单 6-15 所示，fflush 函数以流指针为参数并且刷新输出缓冲区。

程序清单 6-15　用 fflush 函数刷新缓冲区

```
char data[] = { 'h', 'e', 'l', 'l', 'o', '\n' };
size_t count = sizeof(data) / sizeof(data[0]);

/* 向流中写数据. */
fwrite(data, sizeof(char), count, stream);

/* 刷新输出缓冲区. */
if (EOF == fflush(stream))
{
    /* 清空缓冲区时产生错误. */
}
```

如果缓冲区不能写入实际的文件，fflush 函数返回 EOF；否则返回 0。

6.4.5　流的读取

和写入相似，流 I/O 为流的读取提供了 4 个函数。

1. 从流中读取数据块

可以用 fread 函数从流中读取数据块。

```
size_t fread(void* data, size_t size, size_t count, FILE* stream);
```

如程序清单 6-16 所示，fread 函数从给定的流 stream 中读取 count 个 size 大小的元素并放入缓冲区 data 中，它返回实际读取的元素个数。

程序清单 6-16　从流中读取 4 个字符的块数据

```
char buffer[5];
size_t count = 4;

/* 从流中读取 4 个字符. */
if (count ! = fread(buffer, sizeof(char), count, stream))
{
    /* occurred 从流中读数据时产生错误. */
}
else
{
    /*以空结尾. */
    buffer[4] = NULL;

    /* 输出缓冲区. */
    MY_LOG_INFO("read: %s", buffer);
}
```

在成功的情况下，返回的元素个数应该等于传递给 count 的值。

2. 从流中读取字符序列

可以用 fgets 函数从给定的流中读取以换行符结尾的字符序列。

```
char* fgets(char* buffer, int count, FILE* stream);
```

如程序清单 6-17 所示，fgets 函数从给定的流 stream 中最多读取 count-1 个字符再加上换行符，并将新行的字符内容放入字符数组 buffer 中。

程序清单 6-17　读取一个换行符结尾的字符序列

```
char buffer[1024];

/* 从流中读取换行符结尾的字符序列. */
if (NULL == fgets(buffer, 1024, stream))
{
    /* occurred 读取流时产生错误. */
}
else
{
    MY_LOG_INFO("read: %s", buffer);
}
```

成功时返回缓冲区指针，否则返回 NULL 指针。

3. 从流中读取单个字符

fgetc 函数可以从流中读取单个无符号字符。

```
int fgetc(FILE* stream);
```

如程序清单 6-18 所示，fgetc 函数从流中读取单个字符并返回一个整数。

程序清单 6-18　从流中读取单个字符

```
unsigned char ch;
int result;

/* 从流中读取单个字符. */
result = fgetc(stream);
if (EOF == result)
{
    /* occurred 从流中读取时产生错误. */
}
else
{
    /* 获取实际字符. */
    ch = (unsigned char) result;
}
```

如果设置了流的文件结束指示符，则返回 EOF。

4. 从流中读取格式数据

可以用 fscanf 函数从流中读取格式数据。它的工作方式和 fprintf 函数相似，不过它根据给定的格式读取数据并放入提供的参数中。

```
int fscanf(FILE* stream, const char* format, ...);
```

函数的参数包括指向流的指针、格式字符串和格式字符串中指定的可变数量参数。格式字符串由普通字符和格式说明符混合而成。其中的普通字符用来指定必须出现在输入中的字符；格式说明符使 fscanf 函数读取并将数据放在给定的参数中，使用最频繁的说明符有：

- **%d、%i：**读取一个有符号十进制数
- **%u：**读取一个无符号十进制数
- **%o：**读取一个八进制数无符号整数
- **%x：**读取一个十六进制数无符号整数
- **%c：**读取单个字符
- **%f：**读取一个浮点数
- **%e：**读取一个固定格式的浮点数
- **%s：**扫描一个字符串
- **%%：**转义%字符

如程序清单 6-19 所示，给 fscanf 函数提供的参数顺序和类型应该与格式字符串中的说明符相匹配。

程序清单 6-19　从流中读取带格式的数据

```
char s[5];
int i;

/*流中有 "The number is 2" */
/* 读带格式的数据. */
if (2 ! = fscanf(stream, "The %s is %d", s, &i))
{
    /* 读带格式的数据时产生错误. */
}
```

若成功，fscanf 函数返回读取的项目个数。若有错误，则返回 EOF。有关格式字符串和所有说明符及其他修改符的列表信息，可在 fscanf 手册中找到：http://pubs.opengroup.org/onlinepubs/009695399/functions/fscanf.html。

5. 检查文件结尾

从流中读取时，如果已经设置了流的文件结束指示符，可以用 feof 函数检查。

```
int feof(FILE* stream);
```

如程序清单 6-20 所示，如果已到达文件结尾，那么 feof 函数会将流指针作为一个参数并返回一个非零值；如果可以从流中读取更多数据，则返回零。

程序清单 6-20　从流中读取字符串直到文件尾

```
char buffer[1024];

/* 直到文件尾. */
while (0 == feof(stream))
{
    /* 读取并输出字符串. */
    fgets(buffer, 1024, stream);
    MY_LOG_INFO("read: %s", buffer);
}
```

6.4.6　搜索位置

可以用 fseek 函数修改流中的位置。

```
int fseek(FILE* stream, long offset, int whence);
```

fseek 函数有三个参数：流指针、相对偏移量和表示相对偏移量参照点的位置参量。位置参量可以取以下三个值：

- **SEEK_SET**：偏移量相对于流的开头。

- **SEEK_CUR**：偏移量相对于当前位置。
- **SEEK_END**：偏移量相对于流结尾。

程序清单 6-21 所示的示例代码写入了 4 个字符，倒回了 4 个流字节且用不同的字符集重写了它们。

程序清单 6-21　倒回 4 个流字节

```
/* 写入流中. */
fputs("abcd", stream);

/* 倒回 4 个字节. */
fseek(stream, -4, SEEK_CUR);

/* 用 efgh 重写了 abcd. */
fputs("efgh", stream);
```

在示例代码中省略了错误检查。如果该操作成功，则 fseek 函数返回零；否则返回非零值表示操作失败。

6.4.7　错误检查

大多数流 I/O 函数返回 EOF 来表示错误并报告文件结尾。如果在之前的操作中发生了错误，则可以用 ferror 函数进行错误检查。

```
int ferror(FILE* stream);
```

如程序清单 6-22 所示，如果给定流的错误标志已被设置为给定流，那么 ferror 函数会返回一个非零值。

程序清单 6-22　检查错误

```
/* 检查错误. */
if (0 ! = ferror(stream))
{
    /*前一次请求中产生错误. */
}
```

6.4.8　关闭流

可以用 fclose 函数关闭流，此时任何缓冲的输出都会被写入流中，而任何缓冲的输入都会被丢弃。

```
int fclose(FILE* stream);
```

fclose 函数以流指针作为参数。如果成功，则返回零，如果在关闭过程中发生错误，则返回 EOF，如程序清单 6-23 所示。

程序清单 6-23　用 fclose 函数关闭流

```
if (0 ! = fclose(stream))
{
    /*关闭流时产生错误. */
}
```

错误有可能表示由于磁盘空间不足，缓冲的输出不能被写入到流中。检查 fclose 函数返回的值是一个很好的做法。

6.5　与进程交互

Bionic 允许原生应用程序启动并与其他原生进程交互。原生代码可以执行 shell 命令；它可以在后台执行一个进程并与之通信，本节将简要介绍其中的一些重要的函数。

6.5.1　执行 shell 命令

可以用 system 函数向 shell 传递命令。为了使用这个函数，应该先包含 stdlib.h 头文件。

```
#include <stdlib.h>
```

如程序清单 6-24 所示，该函数阻塞了原生代码直到命令执行结束。

程序清单 6-24　用系统函数执行 Shell 命令

```
int result;

/* 执行 shell 命令. */
result = system("mkdir /data/data/com.example.hellojni/temp");
if (-1 == result || 127 == result)
{
    /* 执行 shell 命令失败. */
}
```

6.5.2　与子进程通信

system 命令不为原生应用程序提供接收进程的输出或者给运行的进程发送命令的通信通道，命令执行结束前原生代码一直在等待。某些情况下，原生代码和执行的进程间需要一个通信信道。

可以用 popen 函数在父进程和子进程之间打开一个双向通道。为了使用这个函数，首先应该先包含 stdio.h 标准头文件。

```
FILE *popen(const char* command, const char* type);
```

popen 函数把将要执行的命令以及要求的通信信道类型作为参数，返回一个流指针。若出现错误，则返回 NULL。如程序清单 6-25 所示，在本章前面提到过的 I/O 流函数就可

以与一个文件交互的方式和子进程进行通信。

程序清单 6-25　向 ls 命令打开一个通道并且打印输出

```
#include <stdio.h>
...
FILE* stream;

/* 向 ls 命令打开一个只读通道. */
stream = popen("ls", "r");
if (NULL == stream)
{
    MY_LOG_ERROR("Unable to execute the command.");
}
else
{
     char buffer[1024];
     int status;

     /* 从命令输出中读取每一行. */
     while (NULL ! = fgets(buffer, 1024, stream))
     {
         MY_LOG_INFO("read: %s", buffer);
     }

     /* 关闭通道并获取其状态. */
     status = pclose(stream);
     MY_LOG_INFO("process exited with status %d", status);
}
```

注意:
默认情况下，popen 流是完全缓冲的。需要时可以使用 fflush 函数刷新缓冲区。

子进程执行完成后，应该用 pclose 函数将流关闭。

```
int pclose(FILE* stream);
```

它将流指针作为参数并等待子程序终止，最后返回 exit 状态。

6.6　系统配置

Android 平台以简单的键-值对的方式保存系统属性。Bionic 提供了一组函数允许原生应用程序查询系统属性。为了使用这些函数，首先应该包含系统属性头文件。

```
#include <sys/system_properties.h>
```

系统属性头文件声明了必要的结构和函数。每个系统属性都包含不超过 PROP_NAME_MAX 个字符的属性名和不超过 PROP_VALUE_MAX 个字符的属性值。

6.6.1 通过名称获取系统属性值

The __system_property_get 函数可用于根据名字查看系统属性。

```
int __system_property_get(const char* name, char* value);
```

如程序清单 6-26 所示，它将 null 结尾的属性值复制到所提供的值指针并返回值的大小。复制的总字节数不会超过 PROP_VALUE_MAX。

程序清单 6-26 通过名称获取系统属性值

```
char value[PROP_VALUE_MAX];

/* 获取 product model 系统属性. */
if (0 == __system_property_get("ro.product.model", value))
{
    /* 系统属性未找到或值为空. */
}
else
{
    MY_LOG_INFO("product model: %s", value);
}
```

如果属性没定义，返回大小为 0 的值。

6.6.2 通过名称获取系统属性

可以用__system_property_find 函数获取一个指向系统属性的直接指针。

```
const prop_info* __system_property_find(const char* name);
```

它通过名称搜索系统属性，如果找到指定属性，就会返回一个指向它的指针；否则返回 NULL。在系统的生命周期内返回的指针一直有效，且它可以缓存以方便日后查询。如程序清单 6-27 所示，可以用__system_property_read 函数从该指针中获取属性值。

程序清单 6-27 通过名称获取系统属性

```
const prop_info* property;

/* 获取 product model 系统属性. */
property = __system_property_find("ro.product.model");
if (NULL == property)
{
    /* 系统属性未找到. */
}
else
{
     char name[PROP_NAME_MAX];
     char value[PROP_VALUE_MAX];
```

```
        /* 获取系统属性名称和值. */
        if (0 == __system_property_read(property, name, value))
        {
            MY_LOG_INFO("%s is empty.");
        }
        else
        {
            MY_LOG_INFO("%s: %s", name, value);
        }
    }
```

__system_property_read 函数用指向系统属性的指针和另外两个指向返回的系统属性名称和属性值的字符数组指针作参数。

```
int __system_property_read(const prop_info* pi, char* name, char* value);
```

它将以 null 结尾的属性值复制到提供的值指针中，并返回值的大小。复制的字符总数不会超过 PROP_VALUE_MAX。名称参数是可选的，如果提供了一个字符数组，它会将系统属性名称复制到给定的值指针中。复制的字符总数不会超过 PROP_NAME_MAX。

6.7　用户和组

Linux 内核是为多用户平台设计的。虽然 Android 一定是被单个用户使用，但它仍然要利用基于用户的权限模型。

- Android 在虚拟机沙箱里运行应用程序，且在系统上将它们当作不同的用户对待。通过单纯地依赖基于用户的权限模型，Android 可以通过阻止应用程序访问其他应用程序的数据和内存来达到保证系统安全的目的。
- 服务和硬件资源也是通过基于用户的权限模型来保护的。每个资源都有自己的保护组。在应用程序部署的过程中，应用程序请求访问这些资源。如果应用程序不是正确的资源组成员，它就不能访问任何额外的资源。

Bionic 为用户和组信息函数提供基本支持，这些函数大多数都是只有很少功能或者没有功能的存根。本节讲解一些重要函数，为了使用这些函数，应该首先包含 unistd.h 标准头文件。

```
#include <unistd.h>
```

6.7.1　获取应用程序用户和组 ID

每一个安装好的应用程序都从 10 000 开始获取自己的用户 ID 和组 ID。较低的 ID 用于系统服务，可以用 getuid 函数获取当前应用程序的用户 ID，如程序清单 6-28 所示。

程序清单 6-28　用 getuid 函数获取应用程序的用户 ID

```
uid_t uid;
```

```
/* 获取应用程序的用户 ID. */
uid = getuid();

MY_LOG_INFO("Application User ID is %u", uid);
```

和用户 ID 相似，当前应用程序的组 ID 可以用 getgid 函数获取，如程序清单 6-29 所示。

程序清单 6-29　用 getgid 函数获取应用程序的组 ID

```
gid_t gid;

/*获取应用程序的组 ID. */
gid = getgid();

MY_LOG_INFO("Application Group ID is %u", gid);
```

6.7.2　获取应用程序用户名

每一个安装好的应用程序都获取分配给用户的用户名，用户名以“app_”开头，后接应用程序号。例如，有用户 ID 为 10 040 的应用程序的用户名是 app_40。通过 getlogin 函数获取用户名，如程序清单 6-30 所示。

程序清单 6-30　用 getlogin 函数获取应用程序的用户名

```
char* username;

/* 获取应用程序的用户名. */
username = getlogin();

MY_LOG_INFO("Application user name is %s", username);
```

6.8　进程间通信

为了避免拒绝服务攻击和内核资源泄漏，Bionic 不提供对 System V 的进程间通信(inter-process communication，IPC)的支持。虽然不支持 System V IPC，Android 平台结构利用自己独有的 Binder 使用了大量的 IPC。Android 应用程序通过 Binder 接口与系统、服务以及其他应用间进行交互。在本书编写时，Bionic 不提供任何可以使原生应用程序与 Binder 接口交互的官方 API。目前，只能通过 Android Java APIs 访问 Binder 接口。

6.9　小结

本章深入学习了 Bionic，它是 Google 为 Android 操作系统提供的 BSD 标准 C 库的衍生库。我们学习了通过 Bionic 访问到原生应用程序的标准 C 库函数，例如内存管理、标准 I/O、进程控制、系统配置以及用户和组管理函数。本章也提到了 API，Bionic 也为原生应用程序提供多线程和联网 API。我们将会在后面的各章中分别讲解这些 API。

第 7 章

原 生 线 程

线程是让单个进程并发执行多个任务的机制。它是共享同一个父进程的内存和资源的轻量级进程，一个进程可以包括多个并行执行的线程。作为同一个进程的一部分，线程之间可以彼此通信并共享数据。Android 支持 Java 和原生代码中的线程。本章将学习用于将并发编程附加到原生代码中的不同策略和 API，主要包含以下内容：

- Java 与 POSIX 线程
- 线程同步
- 控制线程的生命周期
- 线程优先级及调度策略
- 原生线程与 Java 的交互

7.1 创建线程示例项目

在详细阐述包含多线程的原生代码之前，需要先创建一个简单的示例应用程序作为测试平台。该示例应用程序包括以下内容：

- 支持原生代码的 Android 应用项目
- 一个简单 GUI，用于定义线程数和每个 worker 迭代的次数、启动线程的按钮、运行时显示原生 worker 进度信息的文本视图
- 模仿运行时间较长任务的原生 worker 函数

在学习本章时，我们将扩展该示例应用程序来说明原生代码中的多线程技术和 API。

7.1.1 创建 Android 项目

开始创建一个新的 Android 应用项目。

(1) 打开 Eclipse IDE，在菜单栏中选择 File | New | Other 打开 New 对话框，如图 7-1 所示。

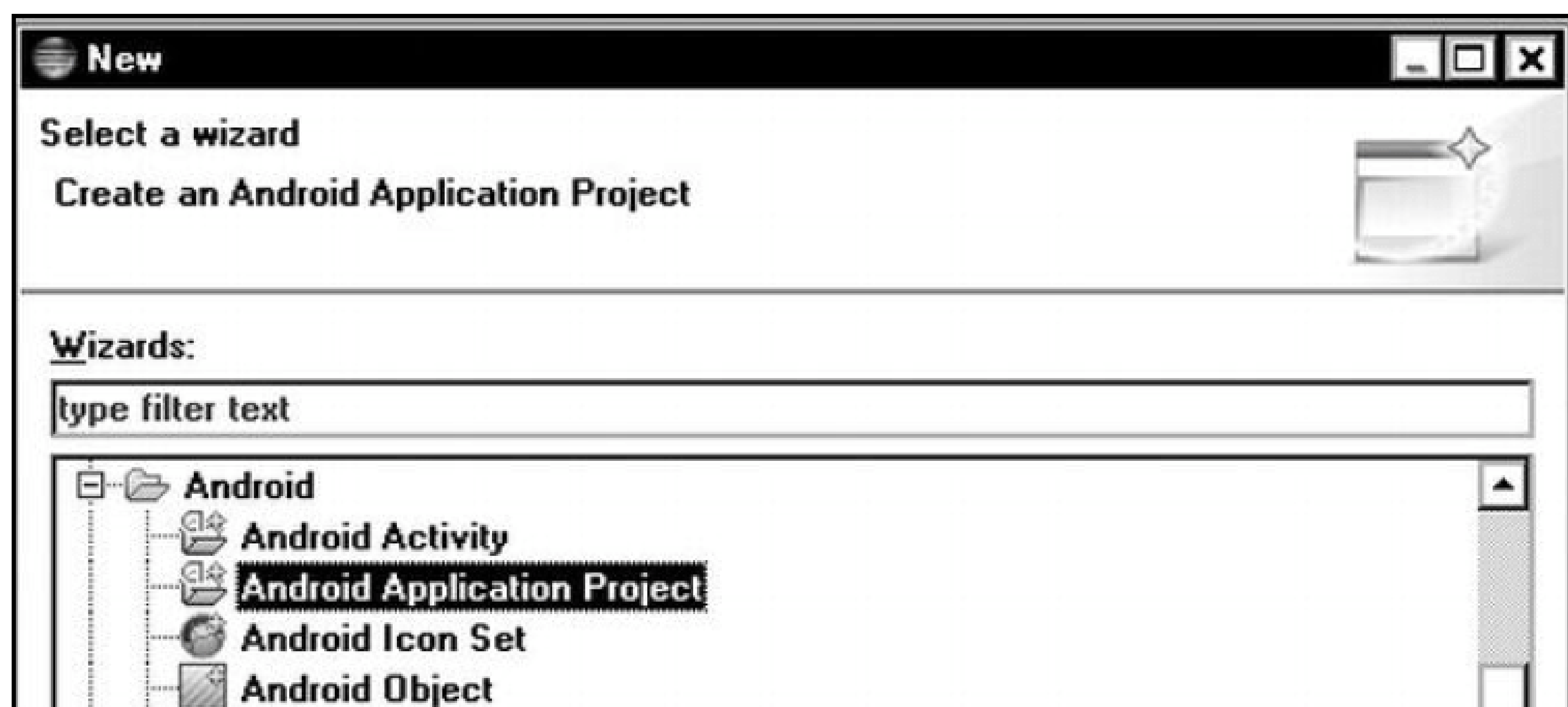


图 7-1　New 对话框

(2) 在向导列表中展开 Android 类别。
(3) 在子列表中选择 Android Application Project。
(4) 单击 Next 按钮打开 New Android App 向导，如图 7-2 所示。

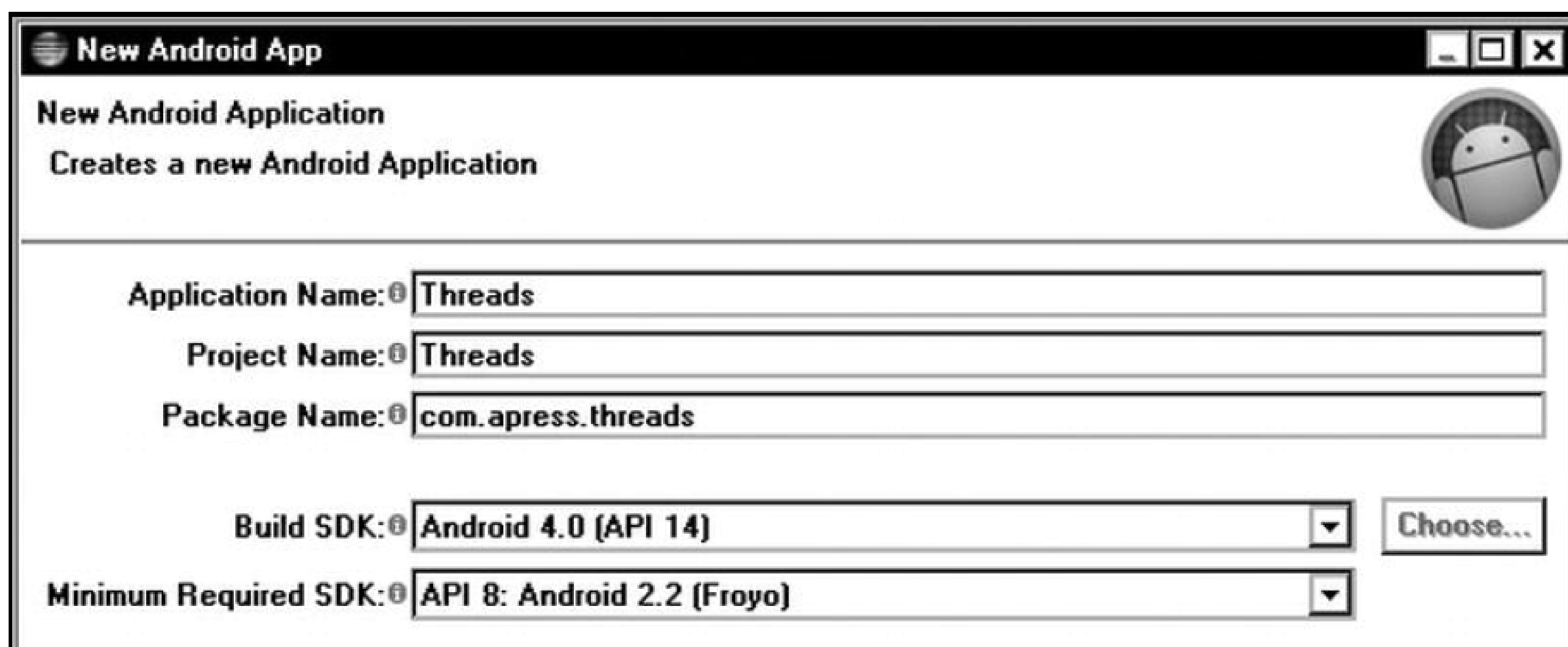


图 7-2　New Android App 对话框

(5) 将 Application Name 设置为 Threads。
(6) 将 Project Name 设置为 Threads。
(7) 将 Package Name 设置为 com.apress.threads。
(8) 将 Build SDK 设置为 Android 4.0。
(9) 将 Minimum Required SDK 设置为 API 8。
(10) 单击 Next 按钮继续。
(11) 单击 Next 按钮让启动图标保持默认设置。
(12) 选择 Create activity。
(13) 在模板列表中选择 Blank Activity。
(14) 单击 Next 继续。
(15) 在 New Blank Activity 步骤中，通过单击 Finish 按钮接受默认值。

7.1.2　添加原生支持

为了使用原生代码，需要将 Native support 添加到新的 Android 项目中。打开 Project Explorer 视图，右击 Threads 项目，并在上下文菜单中选择 Android Tools | Add Native Support。如图 7-3 所示，将弹出 Add Android Native Support 对话框。

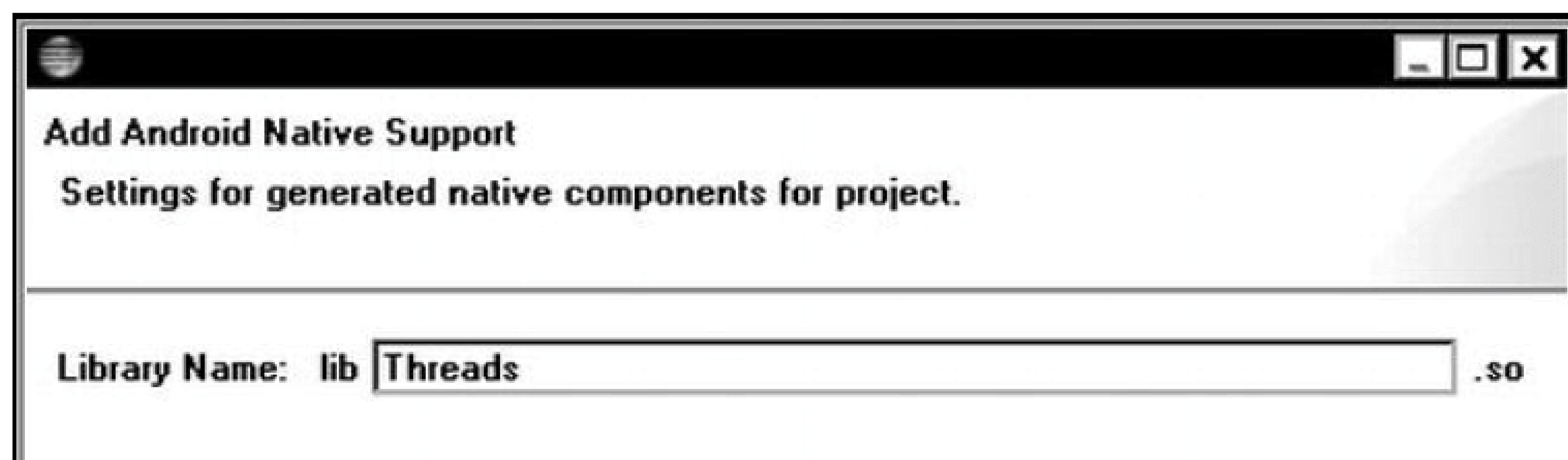


图 7-3　添加 Android Native Support 对话框

将 Library Name 设置为 Threads，并单击 Finish 按钮。Native code support 将被添加到项目中。

7.1.3　声明字符串资源

应用程序的用户接口与一组字符串资源相关。打开 Project Explorer 视图，展开用于保存资源的 res 目录。展开 values 子目录，双击 strings.xml 在编辑器中打开字符串资源。用程序清单 7-1 所示的内容替换字符串资源的内容。

程序清单 7-1　res/values/strings.xml 文件内容

```
<resources>
    <string name="app_name">Threads</string>
    <string name="menu_settings">Settings</string>
    <string name="title_activity_main">Threads</string>
    <string name="threads_edit">Thread Count</string>
    <string name="iterations_edit">Iteration Count</string>
    <string name="start_button">Start Threads</string>
</resources>
```

7.1.4　创建简单的用户界面

用户程序需要有一个简单的用户界面，其中需要包括以下内容：输入线程数和迭代次数的字段、线程启动按钮和监控线程进度的文本视图(如图 7-4 所示)。

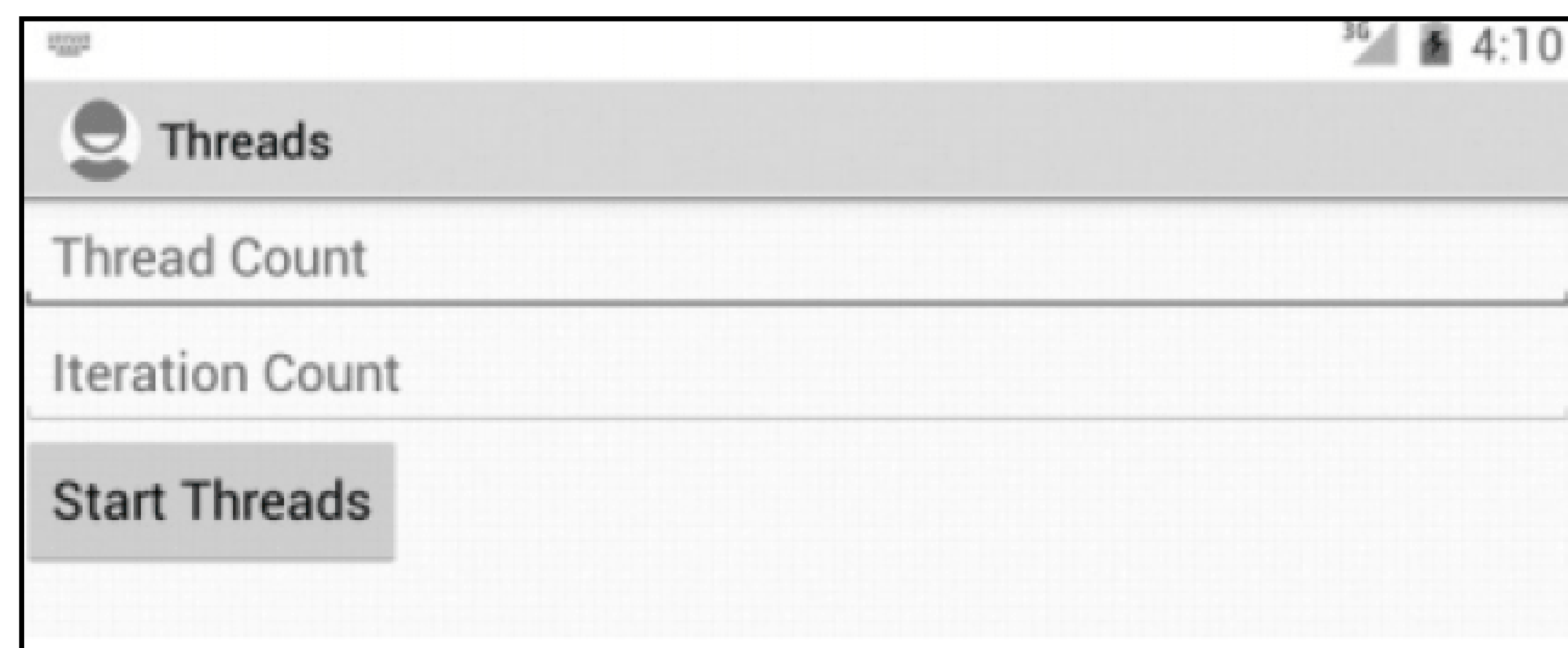


图 7-4　示例应用的简单用户接口

打开 Project Explorer 视图，展开 res 目录下的 layout 子目录。双击 activity_main.xml 布局文件在编辑器中打开它，用程序清单 7-2 所示的内容替换文件原来的内容。

程序清单 7-2　res/layout/activity_main.xml 文件内容

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <EditText
        android:id="@+id/threads_edit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="@string/threads_edit"
        android:inputType="number" >

        <requestFocus />
    </EditText>

    <EditText
        android:id="@+id/iterations_edit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="@string/iterations_edit"
        android:inputType="number" />

    <Button
        android:id="@+id/start_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/start_button" />

    <ScrollView
        android:id="@+id/scrollView1"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content" >

        <TextView
            android:id="@+id/log_view"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

    </ScrollView>

</LinearLayout>
```

7.1.5　实现 Main Activity

main activity 将展示第 7.1.4 节定义的用户接口，它将激活用户接口以便于在运行时配置并控制线程和 workers。在详细介绍 main activity 中提供的函数之前，打开 Project Explorer 视图，展开 src 目录，并且选择 Java 包 com.apress.thread。双击 MainActivity.java 文件，用程序清单 7-3 所示的内容替换文件原来的内容。

程序清单 7-3　src/com/apress/threads/MainActivity.java 文件内容

```
package com.apress.threads;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

/**
 * Main activity.
 *
 * @author Onur Cinar
 */
public class MainActivity extends Activity {
    /** Threads edit. */
    private EditText threadsEdit;

    /** Iterations edit. */
    private EditText iterationsEdit;

    /** Start button. */
    private Button startButton;

    /** Log view. */
    private TextView logView;

    @Override
```

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    // 初始化原生代码
    nativeInit();

    threadsEdit = (EditText) findViewById(R.id.threads_edit);
    iterationsEdit = (EditText) findViewById(R.id.iterations_edit);
    startButton = (Button) findViewById(R.id.start_button);
    logView = (TextView) findViewById(R.id.log_view);

    startButton.setOnClickListener(new OnClickListener() {
        public void onClick(View view) {
            int threads = getNumber(threadsEdit, 0);
            int iterations = getNumber(iterationsEdit, 0);

            if (threads > 0 && iterations > 0) {
                startThreads(threads, iterations);
            }
        }
    });
}

@Override
protected void onDestroy() {
    //释放原生资源
    nativeFree();

    super.onDestroy();
}
/**
 * 原生消息回调.
 *
 * @param message
 *           原生消息.
 */
private void onNativeMessage(final String message) {
    runOnUiThread(new Runnable() {
        public void run() {
            logView.append(message);
            logView.append("\n");
        }
    });
}

/**
 * 以 integer 格式获取编辑文本的值。If the value
 * 如果值为 empty 或计数不能分析，则返回默认值
 *
```

```
     * @param editText edit text.
     * @param defaultValue default value.
     * @return numeric value.
     */
    private static int getNumber(EditText editText, int defaultValue) {
        int value;

        try {
            value = Integer.parseInt(editText.getText().toString());
        } catch (NumberFormatException e) {
            value = defaultValue;
        }

        return value;
    }

    /**
     * 启动给定个数的线程进行迭代   *

     * @param threads thread count.
     * @param iterations iteration count.
     */
    private void startThreads(int threads, int iterations) {
        // 在讲解本章时将实现该方法
    }

    /**
     * 初始化原生代码.
     */
    private native void nativeInit();
    /**
     *释放原生资源.
     */
    private native void nativeFree();
    /**
     * 原生 worker.
     *
     * @param id worker id.
     * @param iterations iteration count.
     */
    private native void nativeWorker(int id, int iterations);

    static {
        System.loadLibrary("Threads");
    }
}
```

除了需要展示和绑定用户接口组件所必需的常用方法之外，main activity 还提供以下主要方法：

- onNativeMessage 是一个由原生代码调用的、用来向 UI 发送进度消息的回调函数。除了主 UI 线程访问并处理 UI 组件之外，Android 不允许代码在不同的线程中运行。因为原生 worker 函数需要在不同的线程中执行，onNativeMessage 方法通过 android.app.Activity 类的 runOnUiThread 方法调度 UI 线程中的实际更新操作。
- startThreads 方法将 start 请求发送到合适的线程示例中。在学习本章时，会涉及线程的不同特性，startThreads 方法将便于在不同的示例之间切换。
- nativeInit 方法在原生代码中实现。在执行线程前完成原生代码的初始化。
- nativeFree 方法在原生代码中实现。当 activity 销毁时该方法负责释放原生资源。
- nativeWorker 方法在原生代码中实现，它模拟执行时间较长的任务，带有两个参数：worked ID 和迭代次数。

7.1.6 生成 C/C++头文件

为了给这两个原生方法生成函数签名，首先打开 Project Explorer，选择 MainActivity.java 源文件，在顶部菜单栏中选择 Run | External Tools | Generate C and C++ Header File。javah 工具将在 jni 目录下生成头文件，内容如程序清单 7-4 所示。

程序清单 7-4 jni/com_apress_threads_MainActivity.h 文件内容

```
/* 不要编辑这个文件 - 它是计算机产生的 */
#include <jni.h>
/* 类 com_apress_threads_MainActivity 的头 */

...

/*
 * Class:     com_apress_threads_MainActivity
 * Method:    nativeInit
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_apress_threads_MainActivity_nativeInit
  (JNIEnv *, jobject);

/*
 * Class:     com_apress_threads_MainActivity
 * Method:    nativeFree
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_apress_threads_MainActivity_nativeFree
  (JNIEnv *, jobject);

/*
 * Class:     com_apress_threads_MainActivity
 * Method:    nativeWorker
 * Signature: (II)V
 */
```

```
JNIEXPORT void JNICALL Java_com_apress_threads_MainActivity_nativeWorker
  (JNIEnv *, jobject, jint, jint);
```

7.1.7 实现原生函数

基于第 7.1.6 节生成的函数签名，现在将实现原生函数。

(1) 打开 Project Explorer，右击 jni 目录。

(2) 在上下文菜单中选择 New | Other 来打开 New 对话框。

(3) 在向导列表中展开 C/C++类别。

(4) 选择 Source File 向导。

(5) 单击 Next 按钮。

(6) 在 New Source File 对话框中，将源文件设置为 com_apress_threads_MainActivity.cpp。

(7) 单击 Finish 按钮。

新建的源文件将在编辑器中打开，用程序清单 7-5 所示内容替换文件原来的内容。

程序清单 7-5 jni/com_apress_threads_MainActivity.cpp 文件的内容

```
#include <stdio.h>
#include <unistd.h>

#include "com_apress_threads_MainActivity.h"

// 方法 ID 能被缓存
static jmethodID gOnNativeMessage = NULL;

void Java_com_après_threads_MainActivity_nativeInit (
        JNIEnv* env,
        jobject obj)
{
    // 如果方法 ID 没被缓存
    if (NULL == gOnNativeMessage)
    {
        // 从对象中获取类
        jclass clazz = env->GetObjectClass(obj);

        // 为回调获取方法 ID
        gOnNativeMessage = env->GetMethodID(clazz,
                "onNativeMessage",
                "(Ljava/lang/String;)V");

        // 如果方法没有找到
        if (NULL == gOnNativeMessage)
        {
            // 获取异常类
            jclass exceptionClazz = env->FindClass(
                    "java/lang/RuntimeException");

            // 抛出异常
```

```
            env->ThrowNew(exceptionClazz, "Unable to find method");
        }
    }
}

void Java_com_apress_threads_MainActivity_nativeFree (
        JNIEnv* env,
        jobject obj)
{

}

void Java_com_apress_threads_MainActivity_nativeWorker (
        JNIEnv* env,
        jobject obj,
        jint id,
        jint iterations)
{
    // 循环给定的迭代数
    for (jint i = 0; i < iterations; i++)
    {
        // 准备消息
        char message[26];
        sprintf(message, "Worker %d: Iteration %d", id, i);

        // 来自 C 字符串的消息
        jstring messageString = env->NewStringUTF(message);

        // 调用原生消息方法
        env->CallVoidMethod(obj, gOnNativeMessage, messageString);

        // 检查是否产生异常
        if (NULL != env->ExceptionOccurred())
            break;

        // 睡眠一秒
        sleep(1);
    }
}
```

原生源文件包含 3 个原生函数：

- **Java_com_apress_threads_MainActivity_nativeInit 函数：**通过找出 onNativeMessage 函数的方法 ID 并将其缓存于全局变量 gOnNativeMessage 中来初始化原生代码。
- **Java_com_apress_threads_MainActivity_nativeFree 函数：**是原生资源释放占位符函数，在本章的学习过程中我们将实现该函数。
- **Java_com_apress_threads_MainActivity_nativeWorker 函数：**用 for 循环模拟运行时间较长的任务，它的循环次数由指定的迭代次数决定，迭代之间休眠 1 秒。通过调用 onNativeMessage 方法将迭代状态传递给 UI。

7.1.8 更新 Android.mk 构建脚本

新建的源文件应该添加到 Android.mk 构建脚本以便于 Android 构建系统将它编译成分享库的一部分。打开 Project Explorer，展开 jni 目录，双击 Android.mk 文件在编辑器中打开。用程序清单 7-6 所示的内容替换文件内容。

程序清单 7-6 jni/Android.mk 文件内容

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE    := Threads
LOCAL_SRC_FILES := com_apress_threads_MainActivity.cpp

include $(BUILD_SHARED_LIBRARY)
```

示例应用程序已经准备完毕，现在可以在 Android 模拟器中运行它以验证示例项目。由于 startThreads 方法还没实现，即使显示 UI，应用程序的功能也不能实现。第 7.2 节将多线程功能添加到示例应用程序中。

7.2 Java 线程

在原生代码中利用多线程的好处的最简单方法就是使用 Java 线程。可以在 Java 空间用纯 Java 代码创建 java.lang.Thread 实例，并在其上下文中调用原生方法。这种方法的主要优点是不要求原生代码做任何修改。

7.2.1 修改示例应用程序使之能够使用 Java 线程

打开 Project Explorer，在编辑器中打开 MainActivity.java 源文件，将 javaThreads 方法添加到 MainActivity 类中，如程序清单 7-7 所示。

程序清单 7-7 将 javaThreads 方法添加到 MainActivity 类

```
public class MainActivity extends Activity {
    ...
    /**
     *使用基于 Java 的线程.
     *
     * @param threads thread count.
     * @param iterations iteration count.
     */
    private void javaThreads(int threads, final int iterations) {
        // 为每一个 worker 创建一个基于 Java 的线程
        for (int i = 0; i < threads; i++) {
            final int id = i;
```

```
                Thread thread = new Thread() {
                    public void run() {
                        nativeWorker(id, iterations);
                    }
                };

                thread.start();
            }
        }
        ...
    }
```

javaThreads 方法有两个参数：线程数和每个 worker 的迭代次数，它完成如下工作：

- 创建请求数目的 java.lang.Thread 对象
- 重载 java.lang.Thread 类的 run 方法以在线程上下文中调用 nativeWorker 方法。
- 启动每一个线程实例。

为了使用 javaThreads 方法，需要修改 startThreads 方法以指向它。 学习本章时，我们要为其他示例重复同样的步骤，这样就可以轻松地在示例之间转换。按照程序清单 7-8 所示更新 startThreads 方法。

程序清单 7-8　修改后的 startThreads 方法调用 javaThreads 方法

```
public class MainActivity extends Activity {
    ...
    /**
     * 启动给定数量的线程迭代.
     *
     * @param threads thread count.
     * @param iterations iteration count.
     */
    private void startThreads(int threads, int iterations) {
        javaThreads(threads, iterations);
    }
    ...
}
```

7.2.2　执行 Java Threads 示例

在 Android 模拟器中运行示例应用程序，步骤如下：

(1) 将线程数设置为 2，使两个线程并发运行。

(2) 将迭代次数设置为 10，使每个线程迭代 10 个步骤。

(3) 单击 Start Threads 按钮启动 Java 线程。

javaThreads 方法将创建两个线程，每个线程将用 10 个迭代运行 nativeWorker 函数，线程将运行 10 秒。开始每一个迭代步骤时，nativeWorker 函数将通过发送一个更新信息来通知 UI，如图 7-5 所示。

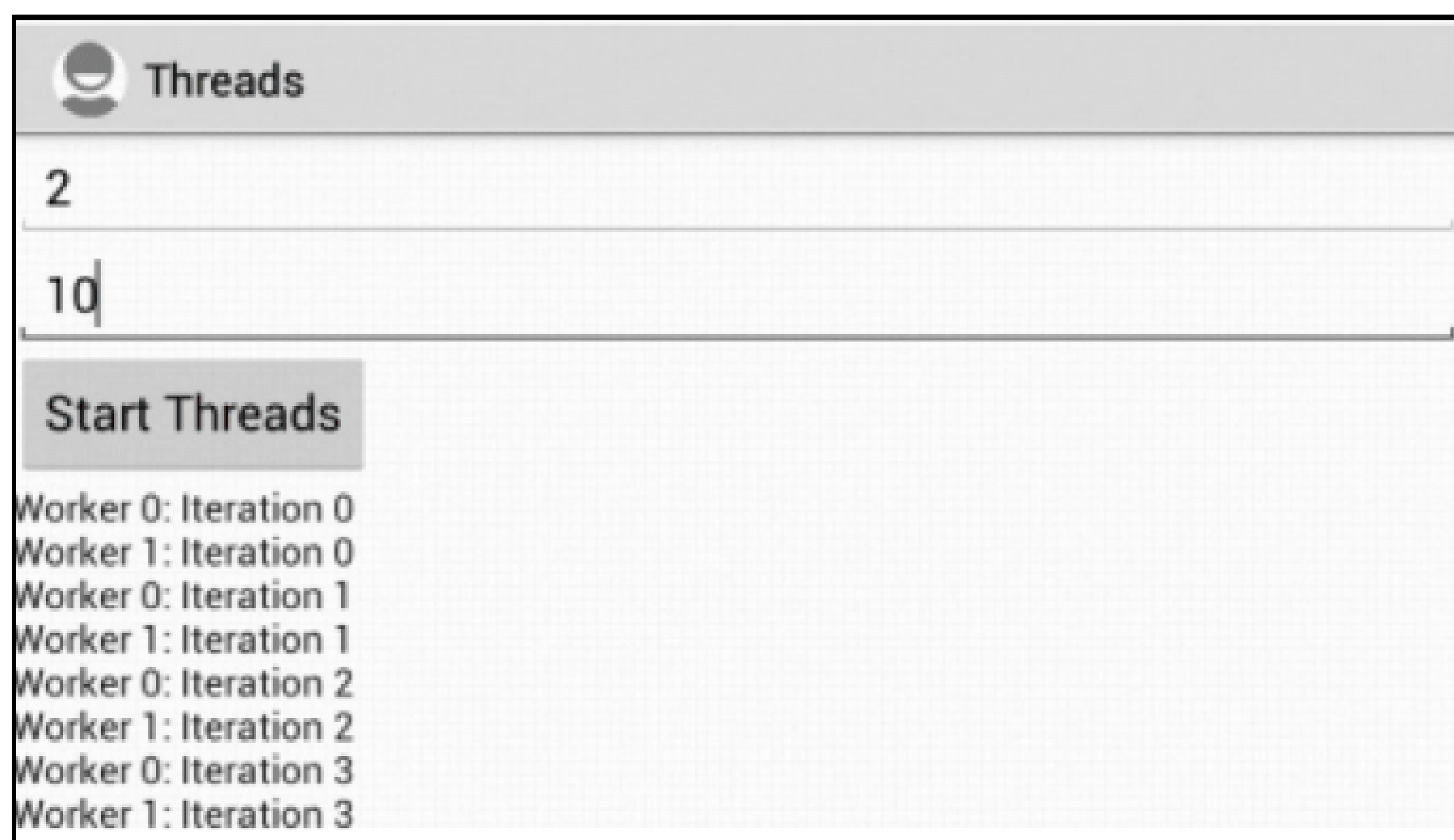


图 7-5　原生代码在 Java 多线程中运行

注意：

如果屏幕过小，你可能需要滚动结果来看最新的更新消息。

7.2.3　原生代码使用 Java 线程的优缺点

与使用原生线程相比，原生代码使用 Java 线程有如下优点：

- 更容易建立。
- 原生代码不要求做任何修改。
- 因为 Java 线程已经是 Java 平台的一部分，所以不要求显式地附着到虚拟机上。原生代码可以用提供的线程专用 JNIEnv 接口指针与 Java 代码通信。
- 通过 java.lang.Thread 类提供的方法可以用于与 Java 代码中的线程实例无缝交互。

虽然有上述优点，但是在用于多线程原生代码时，与使用原生线程相比原生代码使用 Java 线程有如下不足：

- 因为原生空间中没有创建 Java 线程的 API，所以假设为线程分配任务的逻辑是 Java 代码的一部分。
- 因为基于 Java 的线程对原生代码是透明的，所以假定原生代码是线程安全的。
- 原生代码不能获益于其他并发程序的概念或组件，例如信号量等，因为原生空间中没有可供 Java 线程使用的相应 API。
- 在不同的线程中运行的原生代码不能通信或直接共享资源。

注意：

尽管 Java 线程中的一些缺点可以通过使用 JNI 来调用必要的 Java API 来解决，但由于通过 JNI 边界是个非常复杂的操作，所以不提倡使用这个方法。

第 7.3 节开始学习原生线程。

7.3 POSIX 线程

POSIX 线程也被简称为 Pthreads，是一个线程的 POSIX 标准。1995 年之前，有一些不同的线程 API 存在。1995 年发布了 POSIX.1c、线程扩展和标准，并且为创建和处理线程定义了一个通用的 API。许多主流操作系统，包括 Microsoft Windows、Mac OS X、BSD 和 Linux 提供满足 POSIX 线程标准的多线程支持。因为 Android 是基于 Linux 操作系统的，所以它为原生代码提供不一致的 POSIX 线程实现。由于 POSIX 标准非常庞大，本节只介绍 Android 平台完全支持的 API。

7.3.1 在原生代码中使用 POSIX 线程

通过 pthread.h 头文件声明 POSIX Thread APIs，为了在原生代码中使用 POSIX Thread，需要先包含这个头文件。

```
#include <pthread.h>
```

POSIX Thread 的 Android 实现是 Bionic 标准 C 标准库的一部分。与其他平台不同，在编译时不需要链接任何其他的库。

7.3.2 用 pthread_create 创建线程

通过 pthread_create 函数创建 POSIX 线程。

```
int pthread_create(pthread_t* thread,
    pthread_attr_t const* attr,
    void* (*start_routine)(void*),
    void* arg);
```

该函数有如下参数：

- 指向 thread_t 类型变量的指针，函数用该指针返回新线程的句柄。
- 指向 pthread_attr_t 结构的指针形式存在的新线程属性，可以通过该属性指定新线程的栈基址、栈大小、守护大小、调度策略和调度优先级等。本章后面的内容将介绍这些属性中的一部分，如果使用默认值，取值可能为 NULL。
- 指向线程启动程序的函数指针，启动程序函数签名格式如下：

```
void* start_rountine (void* args)
```

启动程序将线程参数看成 void 指针，返回 void 指针类型结果。

当线程以空指针的形式执行时，参数都需要被传递给启动程序，如果不需要传递参数，它可以为 NULL。

成功时，pthread_create 函数返回 0，否则返回一个错误代码。

7.3.3　更新示例应用程序以使用 POSIX 线程

现在可以扩展示例应用程序以使用 POSIX 线程来测试 pthread_create 函数。

1. 更新 Main Activity

打开 Project Explorer，在编辑器中打开 MainActivity.java 源文件。将原生 posixThreads 方法添加到 MainActivity 类中，如程序清单 7-9 所示。

程序清单 7-9　将原生 posixThreads 方法添加到 MainActivity 类

```
public class MainActivity extends Activity {
    ...
    /**
     * 使用 POSIX 线程.
     *
     * @param threads thread count.
     * @param iterations iteration count.
     */
    private native void posixThreads(int threads, int iterations);
    ...
}
```

和 javaThreads 方法相似，posixThreads 方法也有两个参数：线程数和每个 worker 的迭代次数。为了使用 posixThreads 方法，需要修改 startThreads 方法，让该方法指向 posixThreads 方法而不是 javaThreads 方法，按照程序清单 7-10 更新 startThreads 方法。

程序清单 7-10　修改 startThreads 方法调用 posixThreads 方法

```
public class MainActivity extends Activity {
    ...
    /**
     * 启动给定数量的线程进行迭代.
     *
     * @param threads thread count.
     * @param iterations iteration count.
     */
    private void startThreads(int threads, int iterations) {
        posixThreads(threads, iterations);
    }
    ...
}
```

2. 为 posixThreads 方法重新生成 C/C++头文件

如果要使用 POSIX 线程，将会在原生代码而不是 Java 代码中实现 posixThreads 方法。在修改 MainActivity 类时，应该更新 com_apress_threads_MainActivity.h 头文件。打开 Project Explorer 选择 MainActivity.java 源文件，在顶部菜单栏中选择 Run | External Tools | Generate C and C++Header File。更新的头文件将包含 posixThreads 原生方法的函数声明，如程序清

单 7-11 所示。

程序清单 7-11　为 posixThreads 生成函数签名

```
/*
 * Class:     com_apress_threads_MainActivity
 * Method:    posixThreads
 * Signature: (II)V
 */
JNIEXPORT void JNICALL Java_com_apress_threads_MainActivity_posixThreads
  (JNIEnv *, jobject, jint, jint);
```

3. 更新原生代码

现在要更新 POSIX 线程的原生代码。由于 POSIX 线程不是 Java 平台的一部分，为了实现同样的功能需要在原生代码中做许多修改。打开 Project Explore，展开 jni 目录，并双击打开 com_apress_threads_MainActivity.cpp 源文件。然后按照以下步骤操作：

(1) 为了在原生代码中使用 POSIX Thread API，需要在源文件中包括 pthread.h 头文件，如程序清单 7-12 所示。

程序清单 7-12　为 POSIX Thread 包括 pthread.h 头文件

```
#include <stdio.h>
#include <unistd.h>

#include <pthread.h>

#include "com_apress_threads_MainActivity.h"
...
```

(2) 如前所述，在运行一个新线程时，pthread_create 函数能传递一个空指针参数给启动程序。com_apress_threads_nativeWorker 函数需要提供两个任务特定参数，worker ID 和迭代次数。为了给启动程序传递多个参数，需要一个新的结构体将这些参数封装起来。添加 NativeWorkerArgs 结构体定义，如程序清单 7-13 所示。

程序清单 7-13　定义 NativeWorkerArgs 结构体

```
#include "com_apress_threads_MainActivity.h"

// 原生 worker 线程参数
struct NativeWorkerArgs
{
    jint id;
    jint iterations;
};

// 能被缓存的方法 ID
static jmethodID gOnNativeMessage = NULL;
```

(3) 由于 POSIX 线程不是 Java 平台的一部分，因此虚拟机不能识别它们。为了和 Java 空间交互，POSIX 线程应该先将自己附着到虚拟机上。为了使 POSIX 线程正确地附着到虚拟机上，Java 虚拟机接口应该为 POSIX 线程所用。一旦它们附着到虚拟机上，在 POSIX 线程上运行的 worker 代码需要调用 onNativeMessage callback 方法来通知 UI。这需要有一个指向 MainActivity 类的引用。因为是一个原生引用，因此 JNI 方法调用提供的对象引用不能缓存。应该创建并存储一个全局引用供线程使用。如程序清单 7-14 所示添加两个全局变量到原生代码。

程序清单 7-14　全局变量保存 Java VM 接口指针和对象实例的全局引用

```
// 能被缓存的方法 ID
static jmethodID gOnNativeMessage = NULL;

// Java 虚拟机接口指针
static JavaVM* gVm = NULL;

// 对象的全局引用
static jobject gObj = NULL;

void Java_com_apress_threads_MainActivity_nativeInit (
        JNIEnv* env,
        jobject obj)
```

(4) 有许多方法可以在原生代码中获得 Java 虚拟机接口指针，最简单、正确的方式是通过 JNI_OnLoad 函数。当共享库开始加载时虚拟机自动调用该函数。该函数将 Java 虚拟机接口指针作为它的一个参数。如程序清单 7-15 所示，为了将 Java 虚拟机接口指针存储到前面步骤中定义的 gVm 全局变量中，需要在原生代码中添加 JNI_OnLoad 函数。

程序清单 7-15　JNI OnLoad 函数存储 Java 虚拟机接口指针

```
jint JNI_OnLoad (JavaVM* vm, void* reserved)
{
    // 缓存 Java 虚拟机接口指针
    gVm = vm;

    return JNI_VERSION_1_4;
}
```

(5) 为了调用 onNativeMessage 回调方法将更新内容从原生代码传递到 UI，需要用到 MainActivity 类实例的对象引用。如程序清单 7-16 所示，更新 Java_com_apress_threads_MainActivity_nativeInit 方法创建一个线程可用的全局引用。

程序清单 7-16　为对象实例创建一个全局引用

```
void Java_com_apress_threads_MainActivity_nativeInit (
        JNIEnv* env,
        jobject obj)
{
```

```
    // 如果对象的全局引用未设置
    if (NULL == gObj)
    {
        // 为对象创建一个新的全局引用
        gObj = env->NewGlobalRef(obj);

        if (NULL == gObj)
        {
            goto exit;
        }
    }

    // 如果方法 ID 未缓存
    if (NULL == gOnNativeMessage)
    ...

exit:
    return;
}
```

(6) 当不再使用时，需要正确地删除全局引用；否则会发生内存泄露。一旦 activity 停止，按照程序清单 7-17 所示更新 Java_com_ apress_threads_MainActivity_nativeFree 函数以删除全局引用。

程序清单 7-17　更新 nativeFree 方法删除全局引用

```
void Java_com_apress_threads_MainActivity_nativeFree (
        JNIEnv* env,
        jobject obj)
{
    // 如果对象的全局引用未设置
    if (NULL != gObj)
    {
        // 删除全局引用
        env->DeleteGlobalRef(gObj);
        gObj = NULL;
    }
}
...
```

(7) 为了在 POSIX 线程中运行 Java_com_apress_threads_MainActivity_nativeWorker 函数，需要一个中间启动程序将 POSIX 线程正确地附着到 Java 虚拟机上，以获得一个有效的 JNIEnv 接口指针，用适当的一组参数执行原生 worker。如程序清单 7-18 所示，添加 nativeWorkerThread 启动程序。

程序清单 7-18　为原生 Worker 线程添加启动程序

```
static void* nativeWorkerThread (void* args)
{
JNIEnv* env = NULL;
```

```
    // 将当前线程附加到 Java 虚拟机上
    // 并且获得 JNIEnv 接口指针
    if (0 == gVm->AttachCurrentThread(&env, NULL))
    {
        // 获取原生 worker 线程参数
        NativeWorkerArgs* nativeWorkerArgs = (NativeWorkerArgs*) args;

        // 在线程上下文中运行原生 worker
        Java_com_apress_threads_MainActivity_nativeWorker(env,
                gObj,
                nativeWorkerArgs->id,
                nativeWorkerArgs->iterations);

        // 释放原生 worker 线程参数
        delete nativeWorkerArgs;

        // 从 Java 虚拟机中分离当前线程
        gVm->DetachCurrentThread();
    }

    return (void*) 1;
}
```

(8) 由于所有的准备工作均已就绪，Java_com_apress_threads_ MainActivity_posixThreads 函数可以在原生代码中实现。函数将用 pthread_create 函数创建新的线程并提供封装在之前定义的 NativeWorkerArgs 结构体中的 worker 参数。出错时函数会抛出一个 java.lang.RuntimeException 并终止。如程序清单 7-19 所示，添加函数到原生代码。

程序清单 7-19　posixThreads 原生方法的实现

```
void Java_com_apress_threads_MainActivity_posixThreads (
        JNIEnv* env,
        jobject obj,
        jint threads,
        jint iterations)
{
    // 为每一个 worker 创建一个 POSIX 线程
    for (jint i = 0; i < threads; i++)
    {
        // 原生 worker 线程参数
        NativeWorkerArgs* nativeWorkerArgs = new NativeWorkerArgs();
        nativeWorkerArgs->id = i;
        nativeWorkerArgs->iterations = iterations;

        // 线程句柄
        pthread_t thread;
         // 创建一个新线程
        int result = pthread_create(
                &thread,
                NULL,
```

```
                nativeWorkerThread,
                (void*) nativeWorkerArgs);

        if (0 != result)
        {
            // 获取异常类
            jclass exceptionClazz = env->FindClass(
                    "java/lang/RuntimeException");

            // 抛出异常
            env->ThrowNew(exceptionClazz, "Unable to create thread");
        }
    }
}
```

现在 POSIX 线程被集成到原生代码中。

7.3.4 执行 POSIX 线程示例

在 Android 模拟器上运行示例应用程序，步骤与为 Java 线程提供的、用 POSIX 线程测试应用程序的步骤相同。尽管底层线程机制是不同的，但应用程序的执行过程是一样的。

7.4 从 POSIX 线程返回结果

当线程终止时，能返回一个结果。这是通过线程启动程序返回的空指针实现。在前面的例子中，Java_com_apress_threads_MainActivity_posixThreads 函数被设计成在线程执行后立即返回。可以将该函数修改成等待线程结束后再返回。通过 pthread_join 函数可以使一个函数等待线程终止。

```
int pthread_join(pthread_t thread, void** ret_val);
```

pthread_ join 函数带有下列参数：

- 线程句柄，它是 pthread_create 函数返回的目标线程。
- 指向空指针的指针，该指针是为了从启动程序中获得返回值。

它将挂起调用线程的执行，直到目标线程终止。如果 ret_val 不是 NULL，该函数将 ret val 指针的值设置为启动程序的返回结果。如果成功，pthread_join 函数返回值是 0；否则它将返回错误代码。

更新原生代码以使用 pthread_ join

为了在 Activity 中看到 pthread_ join，需要更新示例应用程序。打开 Project Explorer，展开 jni 目录，双击 com_apress_threads_ MainActivity.cpp 源文件在编辑器中打开它。如程序清单 7-20 所示更新 Java_com_apress_ threads_MainActivity_posixThreads 函数。

程序清单 7-20　将 pthread_join 添加到原生代码

```
void Java_com_apress_threads_MainActivity_posixThreads (
        JNIEnv* env,
        jobject obj,
        jint threads,
        jint iterations)
{
    // 线程句柄
    pthread_t* handles = new pthread_t[threads];

    // 为每个 worker 创建一个 POSIX 线程
    for (jint i = 0; i < threads; i++)
    {
        // 原生 worker 线程参数
        NativeWorkerArgs* nativeWorkerArgs = new NativeWorkerArgs();
        nativeWorkerArgs->id = i;
        nativeWorkerArgs->iterations = iterations;

        // 创建新线程
        int result = pthread_create(
                  &handles[i],
                NULL,
                nativeWorkerThread,
                (void*) nativeWorkerArgs);

        if (0 != result)
        {
            // 获取异常类
            jclass exceptionClazz = env->FindClass(
                    "java/lang/RuntimeException");

            // 抛出异常
            env->ThrowNew(exceptionClazz, "Unable to create thread");
             goto exit;
        }
    }

    // 等待线程终止
    for (jint i = 0; i < threads; i++)
    {
        void* result = NULL;

        // 连接每个线程句柄
        if (0 != pthread_join(handles[i], &result))
        {
            //获取异常类
            jclass exceptionClazz = env->FindClass(
                    "java/lang/RuntimeException");
```

```
                // 抛出异常
                env->ThrowNew(exceptionClazz, "Unable to join thread");
            }
            else
            {
                // 准备 message
                char message[26];
                sprintf(message, "Worker %d returned %d", i, result);

                // 来自 C 字符串的 Message
                jstring messageString = env->NewStringUTF(message);

                // 调用原生消息方法
                env->CallVoidMethod(obj, gOnNativeMessage, messageString);

                // 检查是否产生异常
                if (NULL != env->ExceptionOccurred())
                {
                    goto exit;
                }
            }
        }

    exit:
        return;
    }
```

在 Android 模拟器上运行示例应用程序要做必要的改变。将线程数和迭代次数设置为一个较小的数字，比如 2，并单击 Start Threads 按钮，你会立即发现 UI 将挂起几秒。这是由于 pthread_join 函数将 UI 的主线程挂起直到创建的线程终止。UI 将显示线程的返回结果。

7.5 POSIX 线程同步

由于运行在相同的进程空间，线程共享相同的内存和资源。这使线程彼此通信和共享数据变得容易，但是有可能产生两种错误：由于并发修改共享资源产生线程干扰和内存不一致性，此时线程同步变得至关重要。线程同步机制确保两个并发运行的线程不同时执行代码的特定部分。和 Java 线程相似，POSIX 线程 API 也提供同步功能。本章主要学习 POSIX 线程提供的两个最常用的同步机制：

- 互斥锁(Mutexes)确保代码的互斥执行，即代码的特定部分不同时执行。
- 信号量(Semaphores)控制对特定数目可用资源的访问，如果没有可用资源，调用线程只是在信号量所涉及的资源上等待，直到资源可用。

7.5.1 用互斥锁同步 POSIX 线程

POSIX 线程 API 通过 pthread_mutex_t 数据类型展示互斥锁到原生代码。POSIX 线程

API从原生代码提供一组交互功能的互斥。使用前，互斥变量应该先被初始化。

1. 初始化互斥锁

POSIX线程API提供两种初始化互斥锁的方法：pthread_mutex_init函数和PTHREAD_MUTEX_INITIALIZER宏。pthread_mutex_init函数将被用来初始化互斥锁。

```
int pthread_mutex_init(pthread_mutex_t* mutex,const pthread_mutexattr_t*
attr);
```

pthread_mutex_init函数有两个参数，一个指向要初始化的互斥变量的指针和一个指向为互斥锁定义属性的pthread_mutextattr_t结构体的指针。如果第二个参数设置为NULL，将使用默认属性。如果默认属性够用，PTHREAD_MUTEX_INITIALIZER宏比pthread_mutex_init函数更合适。

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

如果初始化成功，互斥锁被初始化并处于锁打开的状态，函数返回 0，否则返回错误代码。

2. 锁定互斥锁

pthread_mutex_lock函数可以通过对一个已经初始化的互斥锁进行封锁操作达到互斥操作的目的。

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
```

该函数带有一个指向互斥锁变量的指针。如果互斥锁已经被锁上，调用线程被挂起直到互斥锁被打开。如果成功，函数返回0，否则返回错误代码。

3. 解锁互斥锁

在临界区代码执行完成时，可使用pthread_mutex_unlock函数解锁互斥锁。

```
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

该函数带有一个指向要解锁的互斥锁变量指针。调度策略决定解锁后执行哪个等待互斥锁的线程。如果成功，函数返回0，否则返回错误代码。

4. 销毁互斥锁

一旦不再需要互斥锁，可以用pthread_mutex_destroy函数销毁互斥锁。该函数带有一个指向要销毁的互斥锁变量的指针，试图销毁一个锁着的变量将返回不确定结果。

```
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

5. 修改示例应用程序以使用互斥锁

现在，修改原生代码以便于实验互斥锁。在Project Explorer中展开jni目录，双击打开com_apress_threads_MainActivity.cpp源文件，按以下步骤操作：

(1) 如程序清单 7-21 所示将互斥锁变量添加到原生代码中。

程序清单 7-21　添加互斥锁到原生代码

```
// 对象的全局引用
static jobject gObj = NULL;

// 互斥实例
static pthread_mutex_t mutex;

jint JNI_OnLoad (JavaVM* vm, void* reserved)
```

(2) 使用之前先初始化互斥锁变量。如程序清单 7-22 所示，更新 Java_com_apress_threads_MainActivity_nativeInit 函数以初始化互斥锁。

程序清单 7-22　初始化互斥锁变量

```
void Java_com_apress_threads_MainActivity_nativeInit (
        JNIEnv* env,
        jobject obj)
{
    // 初始化互斥
    if (0 != pthread_mutex_init(&mutex, NULL))
    {
        // 获取异常类
        jclass exceptionClazz = env->FindClass(
                "java/lang/RuntimeException");

        // 抛出异常
        env->ThrowNew(exceptionClazz, "Unable to initialize mutex");
        goto exit;
    }
    ...
}
```

(3) 一旦不再需要互斥锁，应该将它销毁。如程序清单 7-23 所示更新 Java_com_apress_threads_MainActivity_nativeFree 函数。

程序清单 7-23　销毁互斥锁变量

```
void Java_com_apress_threads_MainActivity_nativeFree (
        JNIEnv* env,
        jobject obj)
{
    ...
    // 销毁互斥锁
    if (0 != pthread_mutex_destroy(&mutex))
    {
        // 获取异常类
        jclass exceptionClazz = env->FindClass(
```

```
                "java/lang/RuntimeException");

        // 抛出异常
        env->ThrowNew(exceptionClazz, "Unable to destroy mutex");
    }
}
```

(4) worker 线程可以在代码开始部分锁定互斥锁并在代码终止部分解锁。如程序清单 7-24 所示，更新 Java_com_apress_threads_ MainActivity_nativeWorker 函数。

程序清单 7-24　锁定和解锁互斥锁

```
void Java_com_apress_threads_MainActivity_nativeWorker (
        JNIEnv* env,
        jobject obj,
        jint id,
        jint iterations)
{
    // 锁定互斥锁
    if (0 != pthread_mutex_lock(&mutex))
    {
        // 获取异常类
        jclass exceptionClazz = env->FindClass(
                "java/lang/RuntimeException");

        //抛出异常
        env->ThrowNew(exceptionClazz, "Unable to lock mutex");
        goto exit;
    }

    ...

    // 解锁互斥锁
    if (0 != pthread_mutex_unlock(&mutex))
    {
        // 获取异常类
        jclass exceptionClazz = env->FindClass(
                "java/lang/RuntimeException");

        // 抛出异常
        env->ThrowNew(exceptionClazz, "Unable to unlock mutex");
    }

exit:
    return;
}
```

可以在 Android 模拟器上运行示例应用程序。由于原生代码正在使用互斥锁，线程将不会同时执行。只有有互斥锁的线程会被执行并向 UI 发送更新信息；其他线程将被挂起等待互斥锁可用。

7.5.2 使用信号量同步 POSIX 线程

和其他 POSIX 函数不同，POSIX 信号量在不同的头文件 semaphore.h.中声明。

```
#include <semaphore.h>
```

原生代码中看到的 POSIX 信号量是 sem_t 类型的。POSIX Semaphore API 提供了一组与原生代码中的信号量交互的函数。在使用之前，信号量变量应该先被初始化。

1. 初始化信号量

POSIX Semaphore API 提供了 sem_init 函数来初始化信号量变量。

```
extern int sem_init(sem_t* sem, int pshared, unsigned int value);
```

它有三个参数：一个将被初始化的信号量变量指针、共享标志(flag)及其初始值。如果成功，函数返回 0，否则返回-1。

2. 锁定信号量

一旦信号量被正确地初始化了，线程可以使用 sem_wait 函数减少信号量的数量。

```
extern int sem_wait(sem_t* sem);
```

这个函数有一个信号量变量指针。如果信号量的值大于零，上锁成功，并且信号量的值也会相应递减。如果信号量的值是零，调用线程被挂起，直到另一个线程通过解锁它增加了信号量的值。如果成功，函数返回 0，否则返回-1。

3. 解锁信号量

在临界区代码执行完成时，线程可以使用 sem_post 函数解锁信号量。

```
extern int sem_post(sem_t* sem);
```

当使用 sem_post 函数解锁信号量以后，信号量的值会增加 1。调度策略决定信号量解锁后执行哪个等待线程。如果成功，函数返回 0，否则返回-1。

4. 销毁信号量

一旦不再需要信号量，可以通过 sem_destroy 函数销毁它。

```
extern int sem_destroy(sem_t* sem);
```

该函数有一个将被销毁的信号量变量指针。销毁一个另一个线程正在阻塞的信号量有可能导致未知行为。如果成功，函数返回 0，否则返回-1。

7.6 POSIX 线程的优先级和调度策略

采用优先级调度的线程调度策略是按照某种执行顺序对线程进行排序，本节简要介绍

调度策略和线程优先级。

7.6.1　POSIX 的线程调度策略

POSIX 线程规范要求实现一组调度策略。最经常使用的调度策略如下：

- SCHED_FIFO：先进先出调度策略基于线程进入列表的时间对线程进行排序，也可以基于优先级在列表中移动线程。
- SCHED_RR：循环轮转调度策略是线程执行时间加以限制的 SCHED_FIFO，其目的是避免线程独占可用的 CPU 时间。

这些调度策略常量在 sched.h 头文件中定义。可以在用 pthread_create 函数创建一个新线程时，用线程属性结构 pthread_attr_t 的 sched_policy 域来定义调度策略；也可以在运行时用 pthread_setschedparam 函数定义调度策略。

```
int pthread_setschedparam(pthread_t thid,int poilcy,
struct sched_param const* param);
```

该函数的参数是一个指向目标线程句柄的指针，调度策略及调度策略所需要的参数。

7.6.2　POSIX Thread 优先级

POSIX Thread API 也提供基于调度策略调整线程优先级的函数。可以在用 pthread_create 函数创建一个新线程时，用线程属性结构 pthread_attr_t 的 sched_priority 域来定义调度优先级；也可以在运行时用 pthread_setschedparam 函数在 sched_param 结构体中提供优先级。优先级的最大值和最小值的取值取决于所使用的调度策略，应用程序可以使用 sched_get_priority_max 函数 and sched_get_priority_min 函数查询这些数。

7.7　小结

本章学习了通过 Java 线程和 POSIX 线程的原生空间提供的可能的多线程机制。本章对这些线程机制进行了比较，重点阐述了 POSIX 线程以便于对原生空间提供的线程 APIs 进行快速概述，如与 POSIX 线程相关的同步、优先级和调度等。

第 8 章

POSIX Socket API：面向连接的通信

因为原生代码应用在一个远离用户的孤立环境中执行，所以它们需要一个与父应用或外界进行通信的媒介以提供服务。第 3 章学习了让原生代码与其父 Java 应用进行通信的 JNI 技术。从本章开始，我们将深入探讨在 Bionic 中可用的 POSIX Socket API，它们可以使原生代码与外界直接通信而不必调用 Java 层。

socket 是一个连接的终点，它可以被命名和寻址以在相同机器或网络中不同机器上的应用程序之间传输数据。POSIX Socket API 以前被称为 Berkeley Socket API，是一个高度通用的设计，它使应用能通过同一组 API 函数在各种协议族上进行通信。

本章将简要介绍面向连接的通信 POSIX Socket API，并重点讨论以下与 Android 平台有关的问题。

- POSIX sockets 概述
- sockets 族
- 面向连接的 sockets

在详细介绍面向连接的通信 POSIX Socket API 之前，先创建一个名为 Echo 的简单示例应用。该示例应用将作为一个测试平台，使我们在学习本章及其后两章的内容时能从不同角度更好地理解 socket 编程。

8.1 Echo Socket 示例应用

该示例应用将提供以下内容：

- 一个用来定义配置 socket 所需参数的简单用户界面。
- 进行简单回显服务的服务逻辑，该服务将接收到的字节重复返回给发送者。

- 样板原生代码片段，方便进行原生层的 Android socket 编程。
- 一个面向连接的 socket 通信示例。
- 一个无连接的 socket 通信示例。
- 一个原生 socket 通信示例。

8.1.1 Echo Android 应用项目

打开 Eclipse IDE，按照下列步骤创建 Echo 应用：

(1) 打开 Android Application Project 对话框。

(2) 将 Application Name 设置为 Echo。

(3) 将 Project Name 设置为 Echo。

(4) 将 Package Name 设置为 com.apress.echo。

(5) 将 Build SDK 设置为 Android 4.1。

(6) 将 Minimum Required SDK 设置为 API 8。

(7) 单击 Next 按钮接受其他设置的默认值。

(8) 单击 Next 按钮选择默认图标为启动图标。

(9) 取消选中 Create activity，并且单击 Finish 按钮创建一个空项目。

(10) 进入 Project Explorer 视图，通过 Android Tools 上下文菜单项打开 Android Native Support 向导。

(11) 将 Library Name 设置为 Echo。

(12) 按照向导要求向项目中添加原生支持。

8.1.2 抽象 echo activity

为了方便重用常用功能，需要在定义实际的 activity 之前创建一个抽象 activity 类。在 Project Explorer 视图中展开 src 目录，右击 com.apress.echo 包，在上下文菜单中选择 New | Class。将 Name 设置为 AbstractEchoActivity 并单击 Finish 按钮。在 Editor 视图中添加如程序清单 8-1 所示的类文件的内容。

程序清单 8-1 AbstractEchoActivity.java 类文件的内容

```
package com.apress.echo;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ScrollView;
import android.widget.TextView;
```

```
/**
 * 抽象 echo activity 对象.
 *
 * @author Onur Cinar
 */
public abstract class AbstractEchoActivity extends Activity implements
        OnClickListener {
    /** 端口号. */
    protected EditText portEdit;

    /** 服务按钮. */
    protected Button startButton;

    /** 日志滚动. */
    protected ScrollView logScroll;

    /** 日志视图. */
    protected TextView logView;

    /** 布局 ID. */
    private final int layoutID;

    /**
     * 构造函数.
     *
     * @param layoutID
     * 布局 ID.
     */
    public AbstractEchoActivity(int layoutID) {
        this.layoutID = layoutID;
    }

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(layoutID);

        portEdit = (EditText) findViewById(R.id.port_edit);
        startButton = (Button) findViewById(R.id.start_button);
        logScroll = (ScrollView) findViewById(R.id.log_scroll);
        logView = (TextView) findViewById(R.id.log_view);

        startButton.setOnClickListener(this);
    }

    public void onClick(View view) {
        if (view == startButton) {
            onStartButtonClicked();
        }
    }
```

```
/**
 * 在开始按钮上单击.
 */
protected abstract void onStartButtonClicked();

/**
 * 以整型获取端口号.
 *
 * @return port number or null.
 */
protected Integer getPort() {
    Integer port;

    try {
         port = Integer.valueOf(portEdit.getText().toString());
    } catch (NumberFormatException e) {
         port = null;
    }

    return port;
}

/**
 * 记录给定的消息
 *
 * @param message
 * 日志消息.
 */
protected void logMessage(final String message) {
    runOnUiThread(new Runnable() {
        public void run() {
            logMessageDirect(message);
        }
    });
}

/**
 * 直接记录给定的消息.
 *
 * @param message
 *日志消息.
 */
protected void logMessageDirect(final String message) {
    logView.append(message);
    logView.append("\n");
    logScroll.fullScroll(View.FOCUS_DOWN);
}

/**
 * 抽象异步 echo 任务.
```

```
     */
    protected abstract class AbstractEchoTask extends Thread {
        /** Handler 对象. */
        private final Handler handler;

        /**
         * 构造函数.
         */
        public AbstractEchoTask() {
            handler = new Handler();
        }

        /**
         * 在调用线程中先执行回调.
         */
        protected void onPreExecute() {
            startButton.setEnabled(false);
            logView.setText("");
        }

        public synchronized void start() {
            onPreExecute();
            super.start();
        }

        public void run() {
            onBackground();
            handler.post(new Runnable() {
                public void run() {
                    onPostExecute();
                }
            });
        }

        /**
         * 新线程中的背景回调.
         */
        protected abstract void onBackground();

        /**
         * 在调用线程中后执行回调.
         */
        protected void onPostExecute() {
           startButton.setEnabled(true);
        }
    }

    static {
       System.loadLibrary("Echo");
    }
}
```

除了处理绑定用户界面组件这类的日常任务，AbstractEchoActivity 还提供一个简单的线程实现，以使应用在一个单独的线程而不是 UI 线程中执行网络操作。

8.1.3 echo 应用程序字符串资源

在 Project Explorer 视图中，展开 res 资源目录。展开 values 子目录并双击 strings.xml，在 Editor 视图中打开字符串资源。用程序清单 8-2 所示的内容替换文件内容。

程序清单 8-2 res/values/strings.xml 资源文件的内容

```
<resources>

    <string name="app_name">Echo</string>
    <string name="title_activity_echo_server">Echo Server</string>
    <string name="port_edit">Port Number</string>
    <string name="start_server_button">Start Server</string>
    <string name="title_activity_echo_client">Echo Client</string>
    <string name="ip_edit">IP Address</string>
    <string name="start_client_button">Start Client</string>
    <string name="send_button">Send</string>
    <string name="message_edit">Message</string>
    <string name="title_activity_local_echo">Local Echo</string>
    <string name="local_port_edit">Port Name</string>

</resources>
```

该应用的用户界面布局将参照这些常见的字符串资源。

8.1.4 原生 echo 模块

原生 echo 模块会为 Java 应用程序提供原生 socket 方法的实现。在 Project Explorer 视图中为原生源文件展开 jni 目录，双击 Echo.cpp C++源文件，用程序清单 8-3 所示的一组辅助函数替换其内容，该组辅助函数有助于 socket 通信示例的实现。

程序清单 8-3 jni/Echo.cpp 文件的内容

```
// JNI
#include <jni.h>

// NULL
#include <stdio.h>

// va_list, vsnprintf
#include <stdarg.h>

// errno
#include <errno.h>
```

```
// strerror_r, memset
#include <string.h>

// socket, bind, getsockname, listen, accept, recv, send, connect
#include <sys/types.h>
#include <sys/socket.h>

// sockaddr_un
#include <sys/un.h>

// htons, sockaddr_in
#include <netinet/in.h>

// inet_ntop
#include <arpa/inet.h>

// close, unlink
#include <unistd.h>

// offsetof
#include <stddef.h>

// 最大日志消息长度
#define MAX_LOG_MESSAGE_LENGTH 256

// 最大数据缓冲区大小
#define MAX_BUFFER_SIZE 80

/**
 * 将给定的消息记录到应用程序.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param format message message format and arguments.
 */
 static void LogMessage(
        JNIEnv* env,
        jobject obj,
        const char* format,
        ...)
{
    // 缓存日志方法 ID
    static jmethodID methodID = NULL;

    // 如果方法 ID 未缓存
    if (NULL == methodID)
    {
        // 从对象获取类
        jclass clazz = env->GetObjectClass(obj);
```

```
        // 从给定方法获取方法 ID
        methodID = env->GetMethodID(clazz, "logMessage",
                "(Ljava/lang/String;)V");

        // 释放类引用
        env->DeleteLocalRef(clazz);
    }

    // 如果找到方法
    if (NULL != methodID)
    {
        // 格式化日志消息
        char buffer[MAX_LOG_MESSAGE_LENGTH];

        va_list ap;
        va_start(ap, format);
        vsnprintf(buffer, MAX_LOG_MESSAGE_LENGTH, format, ap);
        va_end(ap);

        // 将缓冲区转换为 Java 字符串
        jstring message = env->NewStringUTF(buffer);

        // 如果字符串构造正确
        if (NULL != message)
        {
            // 记录消息
            env->CallVoidMethod(obj, methodID, message);

            // 释放消息引用
            env->DeleteLocalRef(message);
        }
    }
}

/**
*用给定的异常类和异常消息抛出新的异常
*
* @param env JNIEnv interface.
* @param className class name.
* @param message exception message.
*/
static void ThrowException(
        JNIEnv* env,
        const char* className,
        const char* message)
{
    // 获取异常类
    jclass clazz = env->FindClass(className);
```

```
        //如果异常类未找到
        if (NULL != clazz)
        {
            // 抛出异常
            env->ThrowNew(clazz, message);

            // 释放原生类引用
            env->DeleteLocalRef(clazz);
        }
    }

    /**
     *
     *用给定异常类和基于错误号的错误消息抛出新异常

     * @param env JNIEnv interface.
     * @param className class name.
     * @param errnum error number.
     */
    static void ThrowErrnoException(
            JNIEnv* env,
            const char* className,
            int errnum)
    {
        char buffer[MAX_LOG_MESSAGE_LENGTH];

        // 获取错误号消息
        if (-1 == strerror_r(errnum, buffer, MAX_LOG_MESSAGE_LENGTH))
        {
            strerror_r(errno, buffer, MAX_LOG_MESSAGE_LENGTH);
        }

        // 抛出异常
        ThrowException(env, className, buffer);
    }
```

在学习本章的过程中，需要通过添加各种 socket 函数的实现对这段源代码进行大幅度修改。

8.2　用 TCP sockets 实现面向连接的通信

通过 TCP socket 实现的面向连接的通信为应用程序提供了健壮的、容错的通信介质。该类连接在通信的整个生命周期内维护一个开放的连接，并且透明地处理应用程序中包的校核和错误检查。为了说明用 socket 建立通信和信息交换的过程，需要修改示例 Echo 应用程序使其包含 TCP 服务器和客户端的 activities。

8.2.1 Echo Server Activity 的布局

在 Project Explorer 视图中，展开 res 目录。展开 layout 子目录并创建一个名为 activity_echo_server.xml 的新布局文件。在 Editor 视图中用程序清单 8-4 所示的内容替换文件的内容。

程序清单 8-4　res/layout/activty_echo_server.xml 文件的内容

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
    <EditText
        android:id="@+id/port_edit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:hint="@string/port_edit"
        android:inputType="number" >

        <requestFocus />
    </EditText>

    <Button
        android:id="@+id/start_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="@string/start_server_button" />

    </LinearLayout>

    <ScrollView
        android:id="@+id/log_scroll"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <TextView
            android:id="@+id/log_view"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </ScrollView>

</LinearLayout>
```

Echo Server 提供一个简单的用户界面，该界面用来获取绑定服务器的端口号，同时在运行时显示原生 TCP 服务器的状态更新。

8.2.2　Echo Server Activity

打开 Project Explorer 视图，在 src 目录下创建一个名为 EchoServerActivity.java 的新类文件。在 Editor 视图中添加如程序清单 8-5 所示的文件内容。

程序清单 8-5　EchoServerActivity.java 文件的内容

```
package com.apress.echo;

/**
 * Echo server.
 *
 * @author Onur Cinar
 */
public class EchoServerActivity extends AbstractEchoActivity {
    /**
    * 构造函数.
    */
    public EchoServerActivity() {
        super(R.layout.activity_echo_server);
    }

    protected void onStartButtonClicked() {
        Integer port = getPort();
        if (port != null) {
            ServerTask serverTask = new ServerTask(port);
            serverTask.start();
        }
    }

    /**
     * 根据给定端口启动 TCP 服务器.
     *
     * @param port
     * 端口号.
     * @throws Exception
     */
    private native void nativeStartTcpServer(int port) throws Exception;

    /**
    * 根据给定端口启动 UDP 服务.
    *
    * @param port
    * 端口号.
    * @throws Exception
    */
```

```
    private native void nativeStartUdpServer(int port) throws Exception;

    /**
    * 服务器端任务.
    */
    private class ServerTask extends AbstractEchoTask {
        /** 端口号. */
        private final int port;

        /**
        * 构造函数.
        *
        * @param port
        *端口号.
        */
        public ServerTask(int port) {
            this.port = port;
        }

        protected void onBackground() {
            logMessage("Starting server.");

            try {
                nativeStartTcpServer(port);
            } catch (Exception e) {
                logMessage(e.getMessage());
            }

            logMessage("Server terminated.");
        }
    }
}
```

EchoServerActivity 从用户那里获得必要的参数，并在一个单独的线程中启动 nativeStartTcpServer 函数和原生 TCP 客户端实现。

8.2.3 实现原生 TCP Server

在 Project Explorer 中选择 EchoServerActivity，在 External Tools 菜单中选择 Generate C and C++ Header File 生成原生头文件。在 Project Explorer 中展开 jni 子目录，在编辑器中打开 Echo.cpp 源文件。把光标移到文件开始处，插入程序清单 8-6 所示的 include 语句以包含原生方法声明。

程序清单 8-6 包含 EchoServerActivity 头文件

```
#include "com_apress_echo_EchoServerActivity.h"
```

1. 创建一个 Socket: socket

socket 用一个名为 socket 描述符的整数表示。除了创建 socket 的函数外，Socket API 函数需要有效的 socket 描述符才能正常工作。可以用 socket 函数来创建 socket。

```
int  socket(int domain, int type, int protocol);
```

为了创建新的 socket，socket 函数需要提供以下参数：

- Domain：指定将会产生通信的 socket 域，并且选择将用到的协议族。在本书编写时，Android 平台支持以下协议族：
  - PF_LOCAL：主机内部通信协议族，该协议族使物理上运行在同一台设备上的应用程序可以用 Socket APIs 彼此通信。
  - PF_INET：Internet 第 4 版协议族，该协议族使应用程序可以与网络上其他地方运行的应用程序进行通信。
- Type：指定通信的语义，支持以下几种主要的 socket 类型。
  - SOCK_STREAM：提供使用 TCP 协议的、面向连接的通信 Stream socket 类型。
  - SOCK_DGRAM：提供使用 UDP 协议的、无连接的通信 Datagram socket 类型。
- Protocol：指定将会用到的协议。对于大多数协议族和协议类型来说，只能使用一个协议。为了选择默认协议，该参数可以设为零。

如果创建了合适的 socket，socket 函数返回相关的 socket 描述符，否则返回-1 且全局变量 errno 被相应地设置成错误值。

在 Editer 视图中，将 NewTcpSocket 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 8-7 所示。

程序清单 8-7　NewTcpSocket 原生辅助函数

```
/**
 * 构造新的 TCP socket.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @return socket descriptor.
 * @throws IOException
 */
static int NewTcpSocket(JNIEnv* env, jobject obj)
{
    // 构造 socket
    LogMessage(env, obj, "Constructing a new TCP socket...");
    int tcpSocket = socket(PF_INET, SOCK_STREAM, 0);

    // 检查 socket 构造是否正确
    if (-1 == tcpSocket)
    {
        // 抛出带错误号的异常
        ThrowErrnoException(env, "java/io/IOException", errno);
```

```
    }

    return tcpSocket;
}
```

该辅助函数创建了一个新的 TCP socket，并在失败时抛出一个 java.lang.IOException。

2. 将 socket 与一个地址绑定：bind

当用 socket 函数创建一个 socket 后，该 socket 存在一个 socket 族空间中，且没为该 socket 分配协议地址。为了使客户能够定位到这个 socket 并与之相连，它需要先与一个地址绑定，可以用 bind 函数将 socket 与地址绑定。

```
int bind(int socketDescriptor, const struct sockaddr* address,socklen_t
addressLength);
```

bind 函数需要以下参数将 socket 与地址绑定：

- socket 描述符：指定将绑定到指定地址的 socket 实例
- address：指定 socket 被绑定的协议地址
- address length：指定传递给函数的协议地址结构的大小

不同协议族使用不同的协议地址。PF_INET 协议族使用 sockaddr_in structure 指定协议地址。sockaddr_in 结构的定义见程序清单 8-8。

程序清单 8-8　sockaddr_in 地址结构

```
struct sockaddr_in {
    sa_family_t sin_family;
    unsigned short int sin_port;
    struct in_addr sin_addr;
}
```

如果 socket 正确绑定，bind 函数就返回零；否则返回-1 且 errno 全局变量被设置为相应的错误值。

在 Editor 视图中，将 BindSocketToPort 辅助函数添加到 Echo.cpp 原生模块源文件中，如程序清单 8-9 所示。

程序清单 8-9　BindSocketToPort 原生辅助函数

```
/**
 * 将 socket 绑定到某一端口号.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param sd socket descriptor.
 * @param port port number or zero for random port.
 * @throws IOException
 */
static void BindSocketToPort(
        JNIEnv* env,
```

```
        jobject obj,
        int sd,
        unsigned short port)
{
    struct sockaddr_in address;

    // 绑定 socket 的地址
    memset(&address, 0, sizeof(address));
    address.sin_family = PF_INET;

    // 绑定到所有地址
    address.sin_addr.s_addr = htonl(INADDR_ANY);

    // 将端口转换为网络字节顺序
    address.sin_port = htons(port);

    // 绑定 socket
    LogMessage(env, obj, "Binding to port %hu.", port);
    if (-1 == bind(sd, (struct sockaddr*) &address, sizeof(address)))
    {
        // 抛出带错误号的异常
        ThrowErrnoException(env, "java/io/IOException", errno);
    }
}
```

如果在地址结构中将端口号设置为零，bind 函数会将第一个可用端口号分配给 socket。可以用 getsockname 函数在 socket 中检索到这个端口号。在 Editor 视图中，将 GetSocketPort 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 8-10 所示。

程序清单 8-10　GetSocketPort 原生辅助函数

```
/**
 * 获取当前绑定的端口号 socket.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param sd socket descriptor.
 * @return port number.
 * @throws IOException
 */
static unsigned short GetSocketPort(
        JNIEnv* env,
        jobject obj,
        int sd)
{
    unsigned short port = 0;
    struct sockaddr_in address;
    socklen_t addressLength = sizeof(address);

    // 获取 socket 地址
```

```
        if (-1 == getsockname(sd,
            (struct sockaddr*) &address,
            &addressLength))
        {
            // 抛出带错误号的异常
            ThrowErrnoException(env, "java/io/IOException", errno);
        }
        else
        {
            // 将端口转换为主机字节顺序
            port = ntohs(address.sin_port);

            LogMessage(env, obj, "Binded to random port %hu.", port);
        }

        return port;
    }
```

正如你看到的，端口号不是直接传递到 sockaddr_in 结构中，相反，先用 htons 函数做转换，这是因为主机和网络字节顺序有差别。

3. 网络字节排序

在硬件层上，不同的机器体系结构使用不同的数据排序和表示规则，这被称为机器字节排序或字节序。例如：

- Big-endian 字节顺序首先储存最重要的字节。
- Little-endian 字节顺序首先储存最不重要的字节。

字节排序规则不同的机器不能直接交换数据。为了使字节排序规则不同的机器能在网络上通信，IP 将 big-endian 字节排序声明为官方的数据传输网络字节排序规则。

由于 Java 虚拟机已经在使用 big-endian 字节排序了，这有可能是你第一次听说数据的字节序。Java 应用程序在进行跨网络通信时，不一定要做数据转换。与此相反，因为 Java 虚拟机不执行原生组件，所以它们使用机器字节排序。

- ARM 和 x86 机器结构使用 little-endian 字节排序
- MIPS 机器结构使用 big-endian 字节排序

在网络上通信时，原生代码需要在机器字节排序和网络字节排序间做必要的转换。

socket 库提供了一组便利函数，使原生应用程序可以透明地处理字节排序转换。这些函数通过 sys/endian.h 头文件声明。

```
#include <sys/endian.h>
```

该头文件提供了以下的便利函数：

- htons 函数：将 unsigned short 从主机字节排序转换到网络字节排序。
- ntohs 函数：和 htons 函数相反，将 unsigned short 从网络字节排序转换到主机字节排序。
- htonl 函数：将 unsigned integer 从主机字节排序转换到网络字节排序。

- ntohl 函数：和 htonl 函数相反，将 unsigned integer 从网络字节排序转换到主机字节排序。

采用这些方便的方法是非常有益的，因为这些方法的实现在编译时是基于目标机器体系结构定义的。如果机器字节排序不同于网络字节排序，这些函数与合适的转换函数映射，否则它们不对数据执行任何操作。本章中将会频繁使用这些方便的函数。

让端口与地址绑定不足以让客户端与之相连，应用程序应该显式地启动对输入连接的 socket 的监听。

4. 监听进入的连接：listen

监听 socket 是通过 listen 函数完成的。

```
int listen(int socketDescriptor, int backlog);
```

为了启动对给定 socket 上输入连接的监听，listen 函数需要提供以下参数：

- socket 描述符：指定应用程序想要监听的输入连接 socket 实例。
- backlog：指定保存挂起的输入连接的队列大小。如果应用程序正在忙于为客户服务，其他输入连接就要排队，队列中挂起的连接数的最大值由 backlog 指定。当输入连接达到 backlog 所限定的值时，其他的输入连接将被拒绝。

如果该函数成功，返回零；否则返回-1 且 errno 全局变量被设置为相应的错误。在 Editor 视图中，将 ListenOnSocket 辅助函数追加到 Echo.cpp 原生模块源文件中，如追加内容程序清单 8-11 所示。

程序清单 8-11　ListenOnSocket 原生辅助函数

```
/**
 * 监听指定的待处理连接的 backlog 的 socket，当 backlog 已满时拒绝新的连接.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param sd socket descriptor.
 * @param backlog backlog size.
 * @throws IOException
 */
static void ListenOnSocket(
        JNIEnv* env,
        jobject obj,
        int sd,
        int backlog)
{
    // 监听给定 backlog 的 socket
    LogMessage(env, obj,
            "Listening on socket with a backlog of %d pending connections.",
            backlog);

    if (-1 == listen(sd, backlog))
    {
```

```
            // 抛出带错误号的异常
            ThrowErrnoException(env, "java/io/IOException", errno);
        }
    }
```

通过 listen 函数监听输入连接只是简单地将输入连接放进一个队列里并等待应用程序显式地接受它们。

5. 接受传入连接：accept

accept 函数用来显式地将输入连接从监听队列里取出并接受它。

```
int accept(int socketDescriptor, struct sockaddr* address, socklen_t*
addressLength);
```

accept 函数是一个阻塞函数。如果在监听队列中没有即将到来的输入连接请求，它会使调用进程进入挂起状态，直到有新的输入连接到达。为了接受一个即将到来的输入连接，accept 函数需要提供以下参数：

- socket descriptor：指定应用程序想要从其上接受输入连接的 socket 实例。
- address pointer：提供了一个地址结构，在该结构中填入被连接的客户协议地址。如果应用程序不需要该信息，它可以被设置为 NULL。
- address length pointer：为要填入的连接客户协议地址提供指定大小的内存空间。如果不需要该信息，它可以被设置为 NULL。

如果 accept 请求成功，该函数返回与该连接实例交互时将会用到的客户 socket 描述符；否则，返回-1 且全局变量 errno 被设置为合适的错误值。

在实例应用程序中，我们将获取关于连接客户端的信息并将之显示在 activity 中，在 Editor 视图，将 LogAddress 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 8-12 所示，该源文件将被用来提取和展示必要的信息。

程序清单 8-12　LogAddress 原生辅助函数

```
/**
 * 记录给定地址的 IP 地址和端口号.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param message message text.
 * @param address adress instance.
 * @throws IOException
 */
static void LogAddress(
        JNIEnv* env,
        jobject obj,
        const char* message,
        const struct sockaddr_in* address)
{
    char ip[INET_ADDRSTRLEN];
```

```
        // 将 IP 地址转换为字符串
        if (NULL == inet_ntop(PF_INET,
                &(address->sin_addr),
                ip,
                INET_ADDRSTRLEN))
        {
            // 抛出带错误号的异常
            ThrowErrnoException(env, "java/io/IOException", errno);
        }
        else
        {
            // 将端口转换为主机字节顺序
            unsigned short port = ntohs(address->sin_port);

            // 记录地址
            LogMessage(env, obj, "%s %s:%hu.", message, ip, port);
        }
    }
```

在 Editor 视图中，将程序清单 8-13 所示的 AcceptOnSocket 辅助函数追加到 Echo.cpp 原生模块源文件中。就像之前所提到的，应用程序会用这个函数接收待输入的连接。

程序清单 8-13　AcceptOnSocket 原生辅助函数

```
/**
 * 在给定的 socket 上阻塞和等待进来的客户连接
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param sd socket descriptor.
 * @return client socket.
 * @throws IOException
 */
static int AcceptOnSocket(
        JNIEnv* env,
        jobject obj,
        int sd)
{
    struct sockaddr_in address;
    socklen_t addressLength = sizeof(address);

    // 阻塞和等待进来的客户连接
    // 并且接受它
    LogMessage(env, obj, "Waiting for a client connection...");

    int clientSocket = accept(sd,
            (struct sockaddr*) &address,
            &addressLength);
```

```
    //如果客户 socket 无效
    if (-1 == clientSocket)
    {
        //抛出带错误号的异常
        ThrowErrnoException(env, "java/io/IOException", errno);
    }
    else
    {
        //记录地址
        LogAddress(env, obj, "Client connection from ", &address);
    }

    return clientSocket;
}
```

接收即将到来的连接时，accept 函数返回新的 socket 描述符可用于与客户端交换数据。

6. 从 socket 接收数据：recv

用 recv 函数实现从 socket 接收数据。

```
ssize_t recv(int socketDescriptor, void* buffer, size_t bufferLength, int
flags);
```

recv 函数也是一个阻塞函数。如果没有从给定的 socket 接收到数据，它会使调用进程进入挂起状态，直到接收到可用数据。为了接受即将到来的输入连接，recv 函数需要提供以下参数：

- socket descriptor：指定应用程序想要从中接收数据的 socket 实例。
- buffer pointer：指向内存地址的指针，该内存用来保存从 socket 接收的数据。
- buffer length：指定缓冲区的大小，recv 函数只会向缓冲区中写入该参数指定大小的内容然后返回。
- flags：指定接收所需要的额外标志。

如果 recv 函数成功，它会返回从 socket 那里接收到的字节数；否则返回-1 且全局变量 errno 将被设置为相应的错误。如果该函数返回零，表示 socket 连接失败。在 Editor 视图中，将 ReceiveFromSocket 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 8-14 所示。

程序清单 8-14　ReceiveFromSocket 原生辅助函数

```
/**
 * 阻塞并接收来自 socket 的数据放到缓冲区.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param sd socket descriptor.
 * @param buffer data buffer.
 * @param bufferSize buffer size.
 * @return receive size.
 * @throws IOException
```

```
 */
static ssize_t ReceiveFromSocket(
        JNIEnv* env,
        jobject obj,
        int sd,
        char* buffer,
        size_t bufferSize)
{
    // 阻塞并接收来自 socket 的数据放到缓冲区
    LogMessage(env, obj, "Receiving from the socket...");
    ssize_t recvSize = recv(sd, buffer, bufferSize - 1, 0);

    // 如果接收失败
    if (-1 == recvSize)
    {
        // 抛出带错误号的异常
        ThrowErrnoException(env, "java/io/IOException", errno);
    }
    else
    {
        // 以 NULL 结尾缓冲区形成一个字符串
        buffer[recvSize] = NULL;

        // 如果数据接收成功
        if (recvSize > 0)
        {
            LogMessage(env, obj, "Received %d bytes: %s",
                    recvSize, buffer);
        }
        else
        {
            LogMessage(env, obj, "Client disconnected.");
        }
    }

    return recvSize;
}
```

ReceiveFromSocket 函数用 recv 函数接收数据，这些数据从给定的 socket 写入到给定的缓冲区中。若出现错误将抛出一个 IOException 异常。通过 socket 发送数据也是以类似的方式完成。

7. 向 socket 发送数据：send

向 socket 发送数据是由 send 函数完成的。

```
ssize_t send(int socketDescriptor, void* buffer, size_t bufferLength, int
flags);
```

与 recv 函数一样，send 函数也是一个阻塞函数。如果 socket 在忙着发送数据，它会使

调用进程进入挂起状态直到 socket 可以传输数据。为了接收即将输入的连接，send 函数需要提供以下参数：

- socket descriptor：指定应用程序想要向其发送数据的 socket 实例。
- buffer pointer：指向内存地址的 buffer 指针，该内存是给定的 socket 发送数据的目的地。
- buffer length：指定缓冲区的大小。send 函数只会向缓冲区传输该参数所指定大小的数据然后返回。
- flags：指定发送所需要的额外标志。

如果发送操作成功，send 函数会返回传送的字节数；否则返回-1 且全局变量 errno 将被设置为相应的错误。与 recv 函数一样，如果该函数返回 0，表示 socket 连接失败。在 Editor 视图中，将 SendToSocket 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 8-15 所示。

程序清单 8-15　SendToSocket 原生辅助函数

```
/**
 * 将数据缓冲区发送到 socket.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param sd socket descriptor.
 * @param buffer data buffer.
 * @param bufferSize buffer size.
 * @return sent size.
 * @throws IOException
 */
static ssize_t SendToSocket(
        JNIEnv* env,
        jobject obj,
        int sd,
        const char* buffer,
        size_t bufferSize)
{
    // 将数据缓冲区发送到 socket
    LogMessage(env, obj, "Sending to the socket...");
    ssize_t sentSize = send(sd, buffer, bufferSize, 0);

    // 如果发送失败
    if (-1 == sentSize)
    {
        // 抛出带错误号的异常
        ThrowErrnoException(env, "java/io/IOException", errno);
    }
    else
    {
        if (sentSize > 0)
        {
```

```
            LogMessage(env, obj, "Sent %d bytes: %s", sentSize, buffer);
        }
        else
        {
            LogMessage(env, obj, "Client disconnected.");
        }
    }

    return sentSize;
}
```

SendToSocket 函数用 send 函数从给定的缓冲区向给定的 socket 发送数据。若发送数据时出现错误，会抛出一个 IOException 异常。现在，实现 TCP 服务器流需要的所有辅助函数都已经准备好了。

8. 原生 TCP 服务器方法

nativeStartTcpServer 原生方法是 TCP Echo 应用程序的核心。在 Editor 视图中，将 nativeStartTcpServer 原生方法追加到 Echo.cpp 原生模块源文件中，如程序清单 8-16 所示。

程序清单 8-16　nativeStartTcpServer 原生方法

```
void Java_com_apress_echo_EchoServerActivity_nativeStartTcpServer(
        JNIEnv* env,
        jobject obj,
        jint port)
{
    // 构造新的 TCP socket.
    int serverSocket = NewTcpSocket(env, obj);
    if (NULL == env->ExceptionOccurred())
    {
        // 将 socket 绑定到某端口号
        BindSocketToPort(env, obj, serverSocket, (unsigned short) port);
        if (NULL != env->ExceptionOccurred())
            goto exit;
        // 如果请求了随机端口号
        if (0 == port)
        {
            // 获取当前绑定的端口号 socket
            GetSocketPort(env, obj, serverSocket);
            if (NULL != env->ExceptionOccurred())
                goto exit;
        }

        // 监听有 4 个等待连接的 backlog 的 socket
        ListenOnSocket(env, obj, serverSocket, 4);
        if (NULL != env->ExceptionOccurred())
            goto exit;

        // 接受 socket 的一个客户连接
```

```
            int clientSocket = AcceptOnSocket(env, obj, serverSocket);
            if (NULL != env->ExceptionOccurred())
                goto exit;

            char buffer[MAX_BUFFER_SIZE];
            ssize_t recvSize;
            ssize_t sentSize;

            // 接收并发送回数据
            while (1)
            {
                // 从 socket 中接收
                recvSize = ReceiveFromSocket(env, obj, clientSocket,
                        buffer, MAX_BUFFER_SIZE);

                if ((0 == recvSize) || (NULL != env->ExceptionOccurred()))
                    break;

                // 发送给 socket
                sentSize = SendToSocket(env, obj, clientSocket,
                        buffer, (size_t) recvSize);

                if ((0 == sentSize) || (NULL != env->ExceptionOccurred()))
                    break;
            }

            // 关闭客户端 socket
            close(clientSocket);
        }

    exit:
        if (serverSocket > 0)
        {
            close(serverSocket);
        }
    }
```

通过本节指定的原生辅助函数，在参数提供的端口上打开了一个服务器 socket 并等待输入连接。当输入连接请求到达时，会接收该连接，然后开始在客户 socket 上接收数据并回显传到客户端的字节数。

8.2.4 Echo 客户端 Activity 布局

在 Project Explorer 视图中，展开 resources 下的 res 目录。展开 layout 子目录，创建一个名为 activity_echo_client.xml 的新布局文件。在 Editor 视图中，用程序清单 8-17 所示内容替换文件原来的内容。

程序清单 8-17　res/layout/activity_echo_client.xml 文件的内容

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <EditText
        android:id="@+id/ip_edit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/ip_edit" >

        <requestFocus />

    </EditText>

    <EditText
        android:id="@+id/port_edit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/port_edit"
        android:inputType="number" />

    <EditText
        android:id="@+id/message_edit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/message_edit" />

    <Button
        android:id="@+id/start_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/start_client_button" />

    <ScrollView
        android:id="@+id/log_scroll"
        android:layout_width="match_parent"
        android:layout_height="0dip"
        android:layout_weight="1.0" >

        <TextView
            android:id="@+id/log_view"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </ScrollView>

</LinearLayout>
```

Echo 客户端提供了一个简单的用户接口以获得远程 IP 地址、端口号以及来自用户的消息负载，同时在运行时显示原生 TCP 客户端的更新状态信息。

8.2.5 Echo 客户端 Activity

打开 Project Explorer 视图，在 src 目录下创建名为 EchoClientActivity. java 的新的类文件，在 Editor 视图中添加如程序清单 8-18 所示的文件内容。

程序清单 8-18 EchoClientActivity.java 文件的内容

```
package com.apress.echo;

import android.os.Bundle;
import android.widget.EditText;

/**
 * Echo 客户端.
 *
 * @author Onur Cinar
 */
public class EchoClientActivity extends AbstractEchoActivity {
    /** IP 地址. */
    private EditText ipEdit;

    /** 消息编辑. */
    private EditText messageEdit;

    /**
     * 构造函数.
     */
    public EchoClientActivity() {
        super(R.layout.activity_echo_client);
    }

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        ipEdit = (EditText) findViewById(R.id.ip_edit);
        messageEdit = (EditText) findViewById(R.id.message_edit);
    }

    protected void onStartButtonClicked() {
        String ip = ipEdit.getText().toString();
        Integer port = getPort();
        String message = messageEdit.getText().toString();

        if ((0 != ip.length()) && (port != null)
            && (0 != message.length())) {
            ClientTask clientTask = new ClientTask(ip, port, message);
```

```
            clientTask.start();
        }
    }

    /**
     * 根据给定服务器 IP 地址和端口号启动 TCP 客户端，并且发送给定消息.
     *
     * @param ip
     * IP 地址
     * @param port
     * 端口号
     * @param message
     * 消息文本
     * @throws Exception
     */
    private native void nativeStartTcpClient(String ip, int port,
            String message) throws Exception;

    /**
     * 客户端任务.
     */
    private class ClientTask extends AbstractEchoTask {
        /** 连接的 IP 地址. */
        private final String ip;

        /** 端口号. */
        private final int port;

        /** 发送的消息文本. */
        private final String message;

        /**
         * 构造函数.
         *
         * @param ip
         * 连接的 IP 地址.
         * @param port
         * 连接的端口号.
         * @param message
         * 发送的消息文本
         */
        public ClientTask(String ip, int port, String message) {
            this.ip = ip;
            this.port = port;
            this.message = message;
        }

        protected void onBackground() {
            logMessage("Starting client.");
```

```
                try {
                    nativeStartTcpClient(ip, port, message);
                } catch (Throwable e) {
                    logMessage(e.getMessage());
                }

                logMessage("Client terminated.");
            }
        }
    }
```

EchoClientActivity 从用户那里获得了必要的参数，并在一个单独的线程中打开了 nativeStartTcpClient 函数和 TCP 客户实现。

8.2.6 实现原生 TCP 客户端

在 Project Explorer 视图中选择 EchoClientActivity，然后在 External Tools 菜单中选择 Generate C and C++ Header File 以生成原生头文件。在 Project Explorer 的编辑器中打开 Echo.cpp 源文件。在该源文件顶部插入 include 语句，如程序清单 8-19 所示。

程序清单 8-19 包含 EchoClientActivity 头文件

```
#include "com_apress_echo_EchoClientActivity.h"
```

该头文件包含 nativeStartTcpClient 函数的函数声明。在实现这个函数之前，需要定义一个用来和地址进行连接的辅助函数。

1. 与地址连接：connect

通过提供协议地址来连接 socket 和 server socket，这是由 connect 函数完成的。

```
int connect(int socketDescriptor, const struct sockaddr *address, socklen_t
addressLength);
```

为了接收即将到来的输入连接，connect 函数需要提供以下参数：

- **socket descriptor：** 指定应用程序想要连接协议地址的 socket 实例。
- **address：** 指定 socket 要连接的协议地址。
- **address length：** 指定所提供的地址结构的长度。

如果尝试连接成功，connect 函数返回零；否则返回-1 且将全局变量 errno 设置为相应的错误。在 Editor 视图中，将 ConnectToAddress 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 8-20 所示。

程序清单 8-20 ConnectToAddress 原生辅助函数

```
/**
 * 连接到给定的 IP 地址和给定的端口号.
 *
 * @param env JNIEnv interface.
```

```
 * @param obj object instance.
 * @param sd socket descriptor.
 * @param ip IP address.
 * @param port port number.
 * @throws IOException
 */
static void ConnectToAddress(
        JNIEnv* env,
        jobject obj,
        int sd,
        const char* ip,
        unsigned short port)
{
    // 连接到给定的 IP 地址和给定的端口号
    LogMessage(env, obj, "Connecting to %s:%uh...", ip, port);

    struct sockaddr_in address;

    memset(&address, 0, sizeof(address));
    address.sin_family = PF_INET;

    // 将 IP 地址字符串转换为网络地址
    if (0 == inet_aton(ip, &(address.sin_addr)))
    {
        // 抛出带错误号的异常
        ThrowErrnoException(env, "java/io/IOException", errno);
    }
    else
    {
        // 将端口号转换为网络字节顺序
        address.sin_port = htons(port);
        // 转换为地址
        if (-1 == connect(sd, (const sockaddr*) &address,
            sizeof(address)))
        {
            // 抛出带错误号的异常
            ThrowErrnoException(env, "java/io/IOException", errno);
        }
        else
        {
            LogMessage(env, obj, "Connected.");
        }
    }
}
```

在 socket 与一个协议地址连接时，POSIX Socket API 函数可以被用于在应用程序和 server 之间交换数据。

2. 原生 TCP Client 方法

nativeStartTcpClient 原生方法是 TCP Echo 应用程序的客户端片段。在 Editor 视图中，将 nativeStartTcpClient 原生方法追加到 Echo.cpp 原生模块源文件中，如程序清单 8-21 所示。

程序清单 8-21　nativeStartTcpClient 原生方法

```
void Java_com_apress_echo_EchoClientActivity_nativeStartTcpClient(
        JNIEnv* env,
        jobject obj,
        jstring ip,
        jint port,
        jstring message)
{
    // 构造新的 TCP socket.
    int clientSocket = NewTcpSocket(env, obj);
    if (NULL == env->ExceptionOccurred())
    {
        // 以 C 字符串形式获取 IP 地址
        const char* ipAddress = env->GetStringUTFChars(ip, NULL);
        if (NULL == ipAddress)
            goto exit;

        // 连接到 IP 地址和端口
        ConnectToAddress(env, obj, clientSocket, ipAddress,
                (unsigned short) port);

        // 释放 IP 地址
        env->ReleaseStringUTFChars(ip, ipAddress);

        // 如果连接成功
        if (NULL != env->ExceptionOccurred())
            goto exit;

        // 以 C 字符串形式获取消息
        const char* messageText = env->GetStringUTFChars(message, NULL);
        if (NULL == messageText)
            goto exit;

        // 获取消息大小
        jsize messageSize = env->GetStringUTFLength(message);

        // 发送消息给 socket
        SendToSocket(env, obj, clientSocket, messageText, messageSize);

        // 释放消息文本
        env->ReleaseStringUTFChars(message, messageText);

        // 如果发送未成功
```

```
            if (NULL != env->ExceptionOccurred())
                goto exit;

            char buffer[MAX_BUFFER_SIZE];

            // 从 socket 接收
            ReceiveFromSocket(env, obj, clientSocket, buffer,
                    MAX_BUFFER_SIZE);
        }

    exit:
        if (clientSocket > -1)
        {
            close(clientSocket);
        }
    }
```

通过本节指定的原生辅助函数，打开了一个 socket 并且将它与参数指定的 IP 地址及端口号连接。连接建立起来之后，会通过 socket 发送提供的消息文本、切换到接收模式，并显示从 socket 接收的数据。如果一切顺利，同样的数据应该在 TCP Echo 服务器回显。在执行应用程序之前，Echo TCP 客户和服务器 activities 需要被添加到 Android Manifest 文件中。

8.2.7　更新 Android Manifest

在 Project Explorer 视图中打开 AndroidManifest.xml，用程序清单 8-22 所示内容替换文件原来的内容。

程序清单 8-22　AndroidManifest.xml 文件

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.apress.echo"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="15" />

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".EchoServerActivity"
            android:label="@string/title_activity_echo_server"
            android:launchMode="singleTop" >
```

```
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".EchoClientActivity"
            android:label="@string/title_activity_echo_client"
            android:launchMode="singleTop" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

重新生成 Android 项目以使修改生效。现在示例应用程序现在已经准备好，可以进行测试了。

8.2.8 运行 TCP Sockets 示例

为了测试 TCP Echo 应用程序，需要建立两个 Android Emulator 实例。用同样的设置创建一个新的 Android Emulator 实例。用 Eclipse IDE 在两个单独的 Android Emulator 实例上启动 EchoClientActivity 和 EchoServerActivity。

1. 配置 Echo TCP 服务器

如图 8-1 所示，EchoServerActivity 会提供一个简单的用户界面，用来指定 TCP 服务器在其上接收连接的端口号。

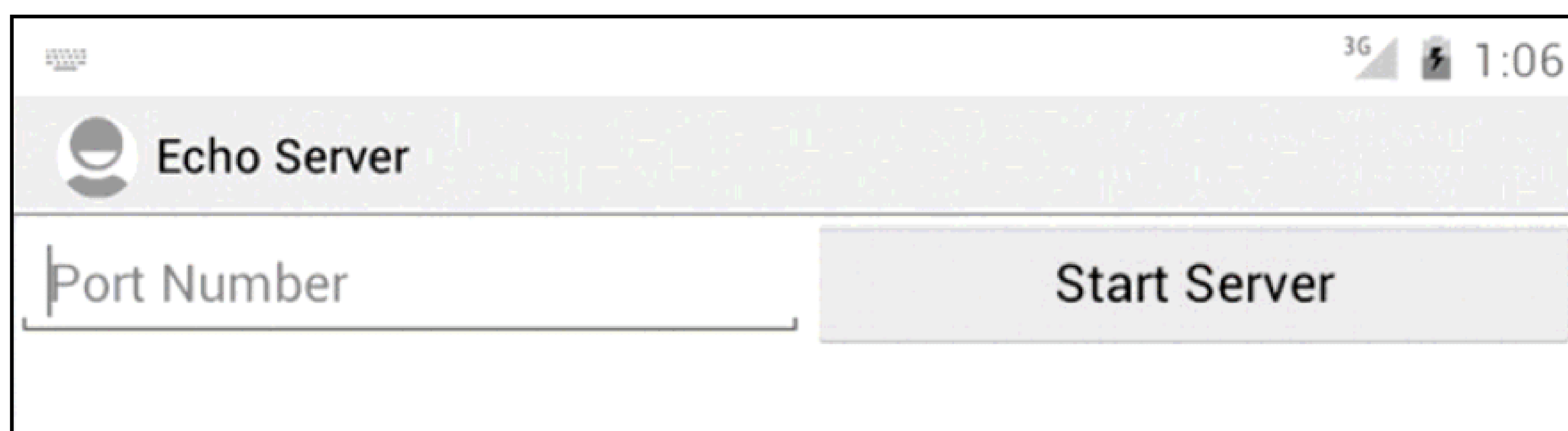


图 8-1　Echo TCP 服务器用户接口

- 将 Port Number 设置为 0。这会从 bind 函数请求分配一个随机端口。
- 单击 Start Server 启动 Echo TCP 服务器。

TCP 服务器开启后，bind 函数会将第一个可用端口号分配给 server socket，且该端口号会在屏幕上显示，如图 8-2 所示。

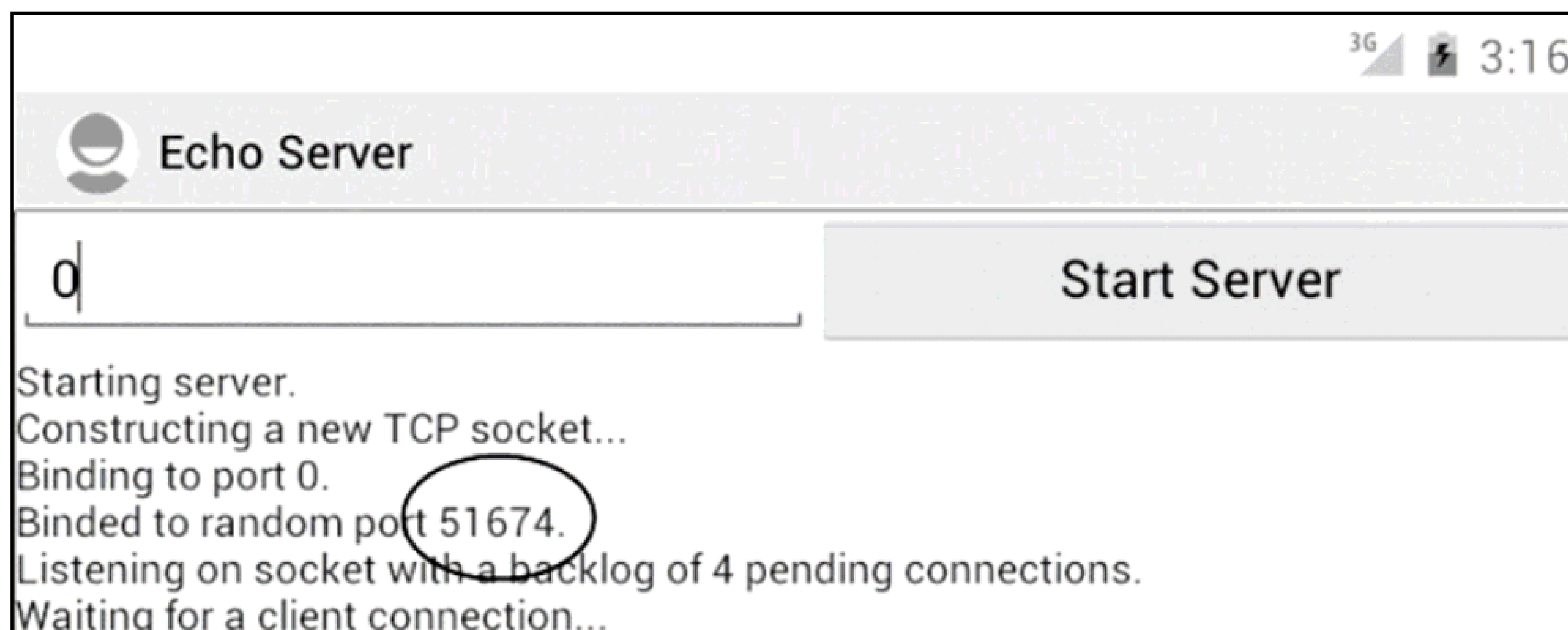


图 8-2　Echo TCP 服务器绑定到一个随机端口号

记录该端口号，因为连接 Echo TCP Client 和 TCP Server 时会用到它。

2. TCP 的连通模拟器

因为 Echo TCP Client 和 TCP Server 都运行在单独的 Android Emulator 实例上，不能直接在它们上建立连接。Android Emulator 作为一个虚拟网络上的虚拟设备，运行在一个沙箱环境中。在 Android Emulator 上运行的应用程序只能和 Android Emulator 进程的宿主机器进行通信。为了使 TCP Client 和 TCP Server 进行通信，TCP 端口号应该通过宿主机来桥接。这可以通过 Android Debug Bridge(adb)提供的端口转发功能来完成。

根据所安装的操作系统，打开一个命令提示符或终端窗口，如程序清单 8-23 所示，用你之前记录的端口号替换<port number>、用 Android Emulator 实例的设备名替换<emulator name>并执行以下命令。

程序清单 8-23　通过 adb 端口转发

```
adb -s <emulator-name> forward tcp:<port number> tcp:<port number>
```

这会将 Android Emulator 上的<port number>与主机上的<port number>映射。任何输入到主机上端口号所指定端口的连接都会通过 adb 转发到 Android Emulator 的指定端口上。端口转发是一种运行时设置，一旦 Android Emulator 停止，它将会被清除。

注意：
如果你正在使用防火墙应用程序，请确保输入连接的端口号是开放的。

3. 配置 Echo TCP 客户端

EchoClientActivity 会提供一个如图 8-3 所示的简单用户界面，它用来指定 IP 地址、要连接的 TCP 服务器的端口号和要传送的信息。

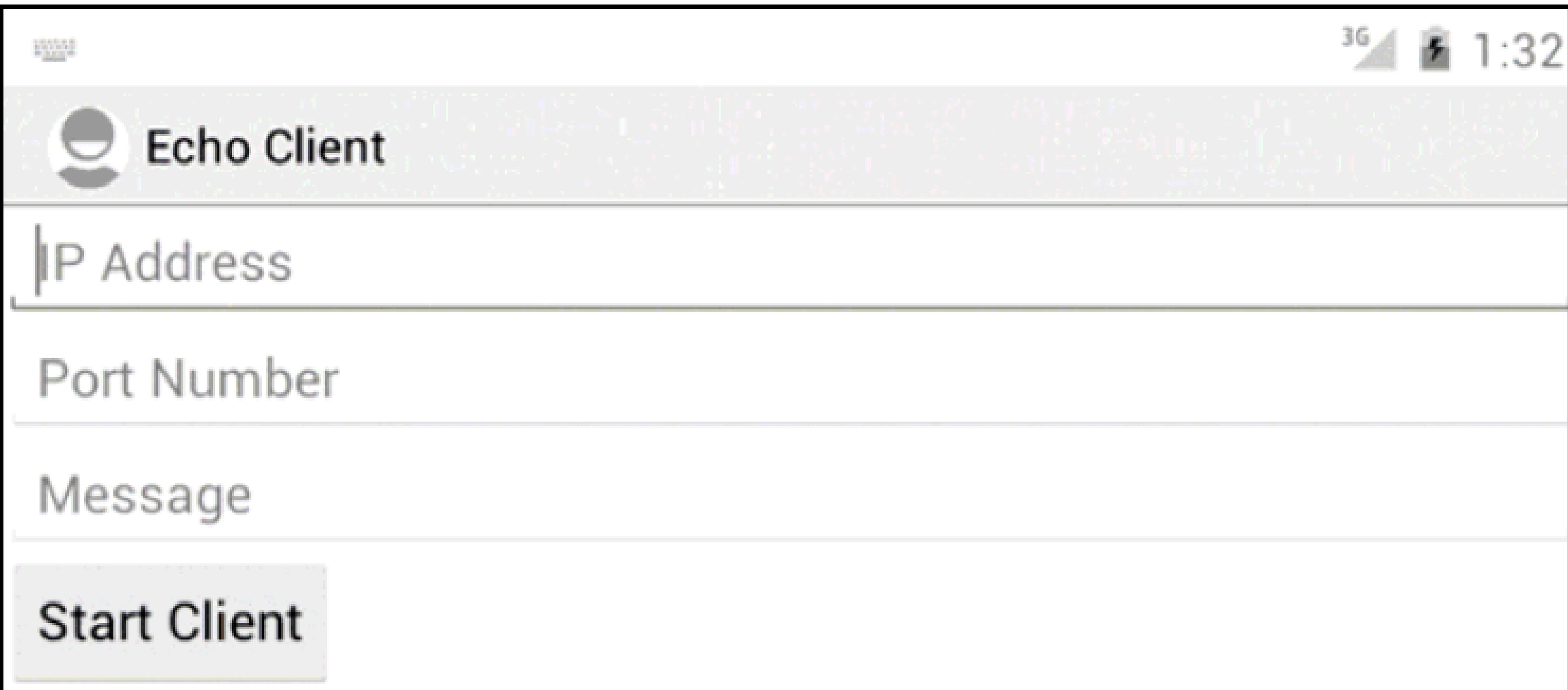


图 8-3　Echo TCP 客户端用户接口

请按照以下步骤启动 TCP echo 客户端应用程序：

(1) 将 IP Address 设置为 10.0.2.2。这是可用来与 Android Emulator 主机进行通信的静态 IP 地址。

(2) 将 Port Number 设置为你之前记录的端口号。

(3) 将 Message 设置为 test 或其他你想要发送给服务器的字符串。

(4) 单击 Start Client 按钮启动 Echo TCP 客户端。

单击 Start Client 按钮后，Echo TCP 客户端会与运行在其他 Android Emulator 实例上的 Echo TCP 服务器连接，并且它会发送消息有效负载。客户端和服务器 activity 都会显示 socket 事件以及被传输的消息，如图 8-4 所示。

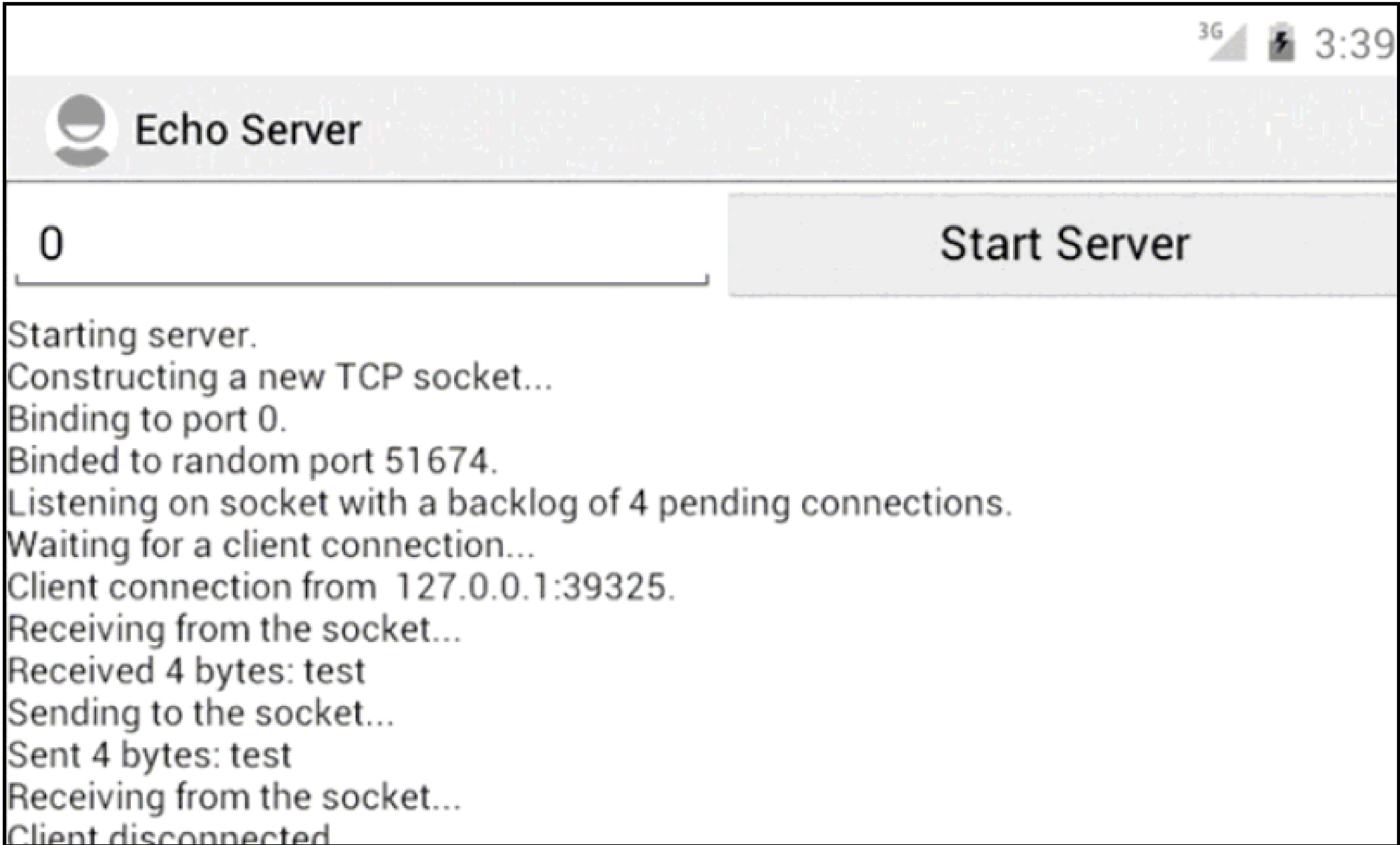


图 8-4　Echo TCP 客户端交换消息

面向连接的协议(如 TCP)为需要可靠通信媒介的应用程序提供无差错的沟通渠道以正常运行。这是以维持一个打开的连接为代价完成的。某些应用程序仍然可以执行而不必保持连接信道，如媒体应用程序。POSIX Socket API 也在原生层提供了对无连接通信的支持。

8.3　小结

本章深入探讨了 Bionic 库为面向连接的通信提供的 POSIX Socket API。学习了使用 TCP 协议的客户端和服务器，本书接下来的两章将继续讨论 POSIX Socket API。第 9 章开始讨论无连接通信的 POSIX Socket API。

第 9 章

POSIX Socket API：无连接的通信

第 8 章通过一个使用 TCP 协议的、面向连接通信的应用程序示例探讨了 POSIX Socket APIs，本章将学习在 Android 应用程序和远端建立无连接通信的方法。无连接通信通过 UDP socket 提供轻量级的通信介质，后面跟着可能解决无序或丢包问题的实时应用程序。该类连接不维护开放的连接。如果需要的话，会将包发送到目标协议地址。因为没有连接，在传输过程中可能出现丢包或无序的情况，该协议不提供处理这种情况的任何服务，本章将继续修改示例应用程序——Echo，使其包含 UDP 服务器和客户端的原生实现。

9.1 将 UDP Server 方法添加到 Echo Server Activity 中

为了用基于 UDP 的 Echo server 进行实验，需要修改 EchoServerActivity 使其包含新的原生方法，如程序清单 9-1 所示。

程序清单 9-1 nativeStartUdpServer 方法添加

```
public class EchoServerActivity extends AbstractEchoActivity {
    ...

    /**
     * 在给定端口上启动 UDP 服务.
     *
     * @param port
     * 端口号.
     * @throws Exception
     */
    private native void nativeStartUdpServer(int port) throws Exception;
```

```
    /**
     * 服务器端任务.
     */
    private class ServerTask extends AbstractEchoTask {
        ...

        protected void onBackground() {
            logMessage("Starting server.");

            try {
                nativeStartUdpServer(port);
            } catch (Exception e) {
                logMessage(e.getMessage());
            }

            logMessage("Server terminated.");
        }
    }
}
```

现在需要在 Echo.cpp 原生源文件中实现该方法。

9.2 实现原生 UDP Server

在 Project Explorer 中选择 EchoServerActivity，再在 External Tools 菜单中选择 Generate C and C++ Header File 以更新生成的原生头文件。

9.2.1 创建 UDP Socket：socket

可以用同一个 socket 函数创建使用 UDP 协议的 socket，这通过让函数创建数据报 socket 而不是流 socket 来实现。为了能够同时用两种类型的连接进行实验，不要修改已有的辅助函数，而是要定义一个新的原生函数以创建 UDP socket。在 Editor 视图中，将 NewUdpSocket 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 9-2 所示。

程序清单 9-2 NewUdpSocket 原生辅助函数

```
/**
 * 构造一个新的 UDP socket.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @return socket descriptor.
 * @throws IOException
 */
static int NewUdpSocket(JNIEnv* env, jobject obj)
{
```

```
        // 构造 socket
        LogMessage(env, obj, "Constructing a new UDP socket...");
        int udpSocket = socket(PF_INET, SOCK_DGRAM, 0);

        // 检查 socekt 构造是否正确
        if (-1 == udpSocket)
        {
            // 抛出带错误号的异常
            ThrowErrnoException(env, "java/io/IOException", errno);
        }

        return udpSocket;
    }
```

NewUdpSocket 是一个创建新数据报 socket 并返回 socket 描述符的简单函数。创建 socket 出错时，socket 函数将返回-1 且全局变量 errno 被设置成相应的错误码。NewUdpSocket 函数抛出一个显示与错误码相对应的错误信息的 IOException，一旦 socket 创建成功，可以用于发送和接收数据报。

9.2.2　从 Socket 接收数据报：recvfrom

用 recvfrom 函数而不是之前用过的 recv 函数实现从 UDP socket 中接收数据。

```
ssize_t recvfrom(int socketDescriptor, void* buffer, size_t bufferLength,
int flags, struct sockaddr* address, socklen_t* addressLength);
```

与 recv 函数一样，recvfrom 函数也是一个阻塞函数。如果没从给定的 socket 接收到数据，它会使调用进程进入挂起状态，直到接收到可用数据。为了接受即将到来的输入连接，recvfrom 函数需要提供以下参数：

- Socket descriptor：指定应用程序想要从中接收数据的 socket 实例。
- buffer pointer：指向内存地址的指针，该内存用来保存从 socket 接收的数据。
- buffer length：指定缓冲区的大小，recvfrom 函数只会向缓冲区中写入该参数指定大小的内容然后返回。
- flags：指定接收所需要的额外标志。
- Address pointer：指定一个地址结构，用于保存客户端发送包的协议地址。如果应用程序不需要该信息，将该参数置为 NULL。
- Address length pointer：指定客户端要写入的协议地址的内存空间大小，如果应用程序不需要该信息，将该参数置为 NULL。

如果 recvfrom 函数成功，将返回从 socket 处接收到的字节数；否则返回-1 且全局变量 errno 将被设置为相应的错误。如果该函数返回零，表示 socket 连接失败。在 Editor 视图中，将 ReceiveDatagramFromSocket 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 9-3 所示。

程序清单 9-3　ReceiveDatagramFromSocket 原生辅助函数

```
/**
 * 从 socket 中阻塞并接收数据报保存到缓冲区，填充客户端地址
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param sd socket descriptor.
 * @param address client address.
 * @param buffer data buffer.
 * @param bufferSize buffer size.
 * @return receive size.
 * @throws IOException
 */
static ssize_t ReceiveDatagramFromSocket(
        JNIEnv* env,
        jobject obj,
        int sd,
        struct sockaddr_in* address,
        char* buffer,
        size_t bufferSize)
{
    socklen_t addressLength = sizeof(struct sockaddr_in);

    // 从 socket 中接收数据报
    LogMessage(env, obj, "Receiving from the socket...");
    ssize_t recvSize = recvfrom(sd, buffer, bufferSize, 0,
            (struct sockaddr*) address,
            &addressLength);

    //如果接收失败
    if (-1 == recvSize)
    {
        // 抛出带错误号的异常
        ThrowErrnoException(env, "java/io/IOException", errno);
    }
    else
    {
        // 记录地址
        LogAddress(env, obj, "Received from", address);

        // 以 NULL 终止缓冲区使其为一个字符串
        buffer[recvSize] = NULL;

        // 如果数据已经接收
        if (recvSize > 0)
        {
            LogMessage(env, obj, "Received %d bytes: %s",
                    recvSize, buffer);
        }
```

```
    }

    return recvSize;
}
```

ReceiveDatagramFromSocket 函数用 recvfrom 函数从给定的 socket 接收数据报，并将接收的数据写入提供的数据缓冲区，出错时则抛出一个 IOException 异常，其中显示相应的错误信息。发送数据报的方式与此类似。

9.2.3　向 Socket 发送数据报：sendto

与 recvfrom 函数一样，给 UDP socket 发送数据也是通过 sendto 函数而不是 send 函数实现。

```
ssize_t sendto(int socketDescriptor, const void* buffer, size_t bufferSize, int flags, const struct sockaddr* address, socklen_t addressLength);
```

与 send 函数一样，sendto 函数也是一个阻塞函数。如果 socket 在忙着发送数据，它会使调用进程进入挂起状态直到 socket 可以传输数据。为了接收即将输入的连接，sendto 函数需要提供以下参数：

- socket descriptor：指定应用程序想要向其发送数据的 socket 实例。
- buffer pointer：指向内存地址的 buffer 指针，该内存是给定的 socket 发送数据的目的地。
- buffer length：指定缓冲区的大小。sendto 函数只会向缓冲区传输该参数所指定大小的数据然后返回。
- flags：指定发送所需要的额外标志。
- Address：指定目标服务器的协议地址。
- Address length：是传递给函数的协议地址结构的大小。

如果发送操作成功，sendto 函数会返回传送的字节数。否则返回-1 且全局变量 errno 将被设置为相应的错误。与 recv 函数一样，如果该函数返回零，表示 socket 连接失败。在 Editor 视图中，将 SendDatagramToSocket 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 9-4 所示。

程序清单 9-4　SendDatagramToSocket 原生辅助函数

```
/**
 * 用给定的 socket 发送数据报到给定的地址.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param sd socket descriptor.
 * @param address remote address.
 * @param buffer data buffer.
 * @param bufferSize buffer size.
 * @return sent size.
```

```
 * @throws IOException
 */
static ssize_t SendDatagramToSocket(
        JNIEnv* env,
        jobject obj,
        int sd,
        const struct sockaddr_in* address,
        const char* buffer,
        size_t bufferSize)
{
    // 向 socket 发送数据缓冲区
    LogAddress(env, obj, "Sending to", address);
    ssize_t sentSize = sendto(sd, buffer, bufferSize, 0,
            (const sockaddr*) address,
            sizeof(struct sockaddr_in));

    // 如果发送失败
    if (-1 == sentSize)
    {
        // 抛出带错误号的异常
        ThrowErrnoException(env, "java/io/IOException", errno);
    }
    else if (sentSize > 0)
    {
        LogMessage(env, obj, "Sent %d bytes: %s", sentSize, buffer);
    }

    return sentSize;
}
```

SendDatagramToSocket 函数用 sendto 函数发送数据，用给定的 socket 以数据报的形式发送数据缓冲区。实现这些辅助函数后，可以开始实现 UDP 服务器函数了。

9.2.4 原生 UDP Server 方法

nativeStartUdpServer 用这些方法提供基于 UDP 的 Echo 服务器。在 Editor 视图中，将 nativeStartUdpServer 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 9-5 所示。

程序清单 9-5 nativeStartUdpServer 原生方法

```
void Java_com_apress_echo_EchoServerActivity_nativeStartUdpServer(
        JNIEnv* env,
        jobject obj,
        jint port)
{
    // 构造一个新的 UDP socket.
    int serverSocket = NewUdpSocket(env, obj);
    if (NULL == env->ExceptionOccurred())
    {
```

```
        // 将 socket 绑定到某一端口号
        BindSocketToPort(env, obj, serverSocket, (unsigned short) port);
        if (NULL != env->ExceptionOccurred())
            goto exit;

        // 如果请求随机端口号
        if (0 == port)
        {
            // 获取当前绑定的端口号 socket
            GetSocketPort(env, obj, serverSocket);
            if (NULL != env->ExceptionOccurred())
                goto exit;
        }

        // 客户端地址
        struct sockaddr_in address;
        memset(&address, 0, sizeof(address));

        char buffer[MAX_BUFFER_SIZE];
        ssize_t recvSize;
        ssize_t sentSize;

        // 从 socket 中接收
        recvSize = ReceiveDatagramFromSocket(env, obj, serverSocket,
                &address, buffer, MAX_BUFFER_SIZE);

        if ((0 == recvSize) || (NULL != env->ExceptionOccurred()))
            goto exit;

        // 发送给 socket
        sentSize = SendDatagramToSocket(env, obj, serverSocket,
                &address, buffer, (size_t) recvSize);
    }

exit:
    if (serverSocket > 0)
    {
        close(serverSocket);
    }
}
```

因为基于 UDP 的服务器是无连接的，所以既不需要监听函数，也不需要接收函数。

9.3　将原生 UDP Client 方法加入 Echo Client Activity 中

为了用基于 UDP 的 Echo 客户端进行实验，需要修改 EchoClientActivity 使其包含一个新的原生方法，如程序清单 9-6 所示。

程序清单 9-6　添加的 nativeStartUdpClient 方法

```
public class EchoClientActivity extends AbstractEchoActivity {
    ...

    /**
     * 用给定的服务器端 IP 地址和端口号启动 UDP 客户端.
     *
     * @param ip
     * IP 地址
     * @param port
     * 端口号
     * @param message
     * 消息文本
     * @throws Exception
     */
    private native void nativeStartUdpClient(String ip, int port,
                                             String message)
            throws Exception;

    /**
     * 客户端任务.
     */
    private class ClientTask extends AbstractEchoTask {
        ...

        protected void onBackground() {
            logMessage("Starting client.");

            try {
                nativeStartUdpClient(ip, port, message);
            } catch (Throwable e) {
                logMessage(e.getMessage());
            }
            logMessage("Client terminated.");
        }
    }
}
```

在将原生方法声明加入 ClientTask 之后，编译应用程序项目生成类文件，现在我们完成了该函数的原生实现。

9.4　实现原生 UDP Client

在 Project Explorer 中选择 EchoClientActivity，然后在 external tools 菜单中选择 Generate C and C++ Header File 以更新生成的原生头文件。

原生 UDP Client 方法

在 Editor 视图中，将 nativeStartUdpClient 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 9-7 所示。

程序清单 9-7　nativeStartUdpClient 原生方法

```
void Java_com_apress_echo_EchoClientActivity_nativeStartUdpClient(
        JNIEnv* env,
        jobject obj,
        jstring ip,
        jint port,
        jstring message)
{
    // 构造一个新的 UDP socket.
    int clientSocket = NewUdpSocket(env, obj);
    if (NULL == env->ExceptionOccurred())
    {
        struct sockaddr_in address;

        memset(&address, 0, sizeof(address));
        address.sin_family = PF_INET;

        // 以 C 字符串形式获取 IP 地址
        const char* ipAddress = env->GetStringUTFChars(ip, NULL);
        if (NULL == ipAddress)
            goto exit;

        // 将 IP 地址字符串转换为网络地址
        int result = inet_aton(ipAddress, &(address.sin_addr));

        // 释放 IP 地址
        env->ReleaseStringUTFChars(ip, ipAddress);

        // 如果转换失败
        if (0 == result)
        {
            // 抛出带错误号的异常
            ThrowErrnoException(env, "java/io/IOException", errno);
            goto exit;
        }

        // 将端口转换为网络字节顺序
        address.sin_port = htons(port);

        // 以 C 字符串形式获取消息
        const char* messageText = env->GetStringUTFChars(message, NULL);
        if (NULL == messageText)
            goto exit;
```

```
            // 获取消息大小
            jsize messageSize = env->GetStringUTFLength(message);

            // 发送消息给 socket
            SendDatagramToSocket(env, obj, clientSocket, &address,
                    messageText, messageSize);

            // 释放消息文本
            env->ReleaseStringUTFChars(message, messageText);

            // 如果发送未成功
            if (NULL != env->ExceptionOccurred())
                goto exit;

            char buffer[MAX_BUFFER_SIZE];

            // 清除地址
            memset(&address, 0, sizeof(address));

            // 从 socket 接收
            ReceiveDatagramFromSocket(env, obj, clientSocket, &address,
                    buffer, MAX_BUFFER_SIZE);
        }

exit:
        if (clientSocket > 0)
        {
            close(clientSocket);
        }
}
```

通过创建一个新的 UDP socket 来启动 nativeStartUdpServer 函数。然后将给定的消息文本作为数据报发送到给定的 IP 地址和端口号。在发送数据报时，开始等待接收响应的数据报。

9.5 运行 UDP Sockets 示例

Echo UDP 服务器和客户端可以用与 Echo TCP 服务器和客户端同样的方式测试，服务器和客户端均运行在两个不同的 Android Emulator 实例上。启动每一个 Echo UDP 服务器时都要将端口号设置为 0，一旦 UDP 服务器启动，立即记录分配的端口号。

9.5.1 连通 UDP 的模拟器

为了给 UDP 端口设置端口转发，需要使用 Android Emulator 控制台。

(1) 首先通过查看 windows 标题确定 Android Emulator 实例的控制台端口号 ，注意是显示在标题栏上的四位数字，例如：5556。

(2) 用你喜欢的 telnet 客户端，连接原生主机及你在步骤(1)记下的端口号。

(3) Android Emulator 控制台上调用下列命令，用每个 UDP Server 的端口号替换<port number>，设置 UDP 端口重定向。

```
redir add udp:<port number>:<port number>
```

注意：

如果你正在使用防火墙应用程序，请确保通过防火墙的端口号是开放的，以接收输入的包。

这会将 Android Emulator 上的 UDP 端口<port number>映射到主机上的 UDP 端口<port number>。任何输入到主机上<port number>所指定端口的连接都会转发到 Android Emulator 的<port number>指定端口上。端口转发是一种运行时设置，一旦 Android Emulator 停止，它将会被清除。

9.5.2　启动 Echo UDP Client

用与配置 Echo TCP Client 相同的参数集配置 Echo UDP client，然后单击 Start Client 按钮。单击 Start Client 按钮后，Echo UDP 客户端会发送消息负载。客户端和服务器 activity 都会显示 socket 事件以及被传输的消息，如图 9-1 所示。

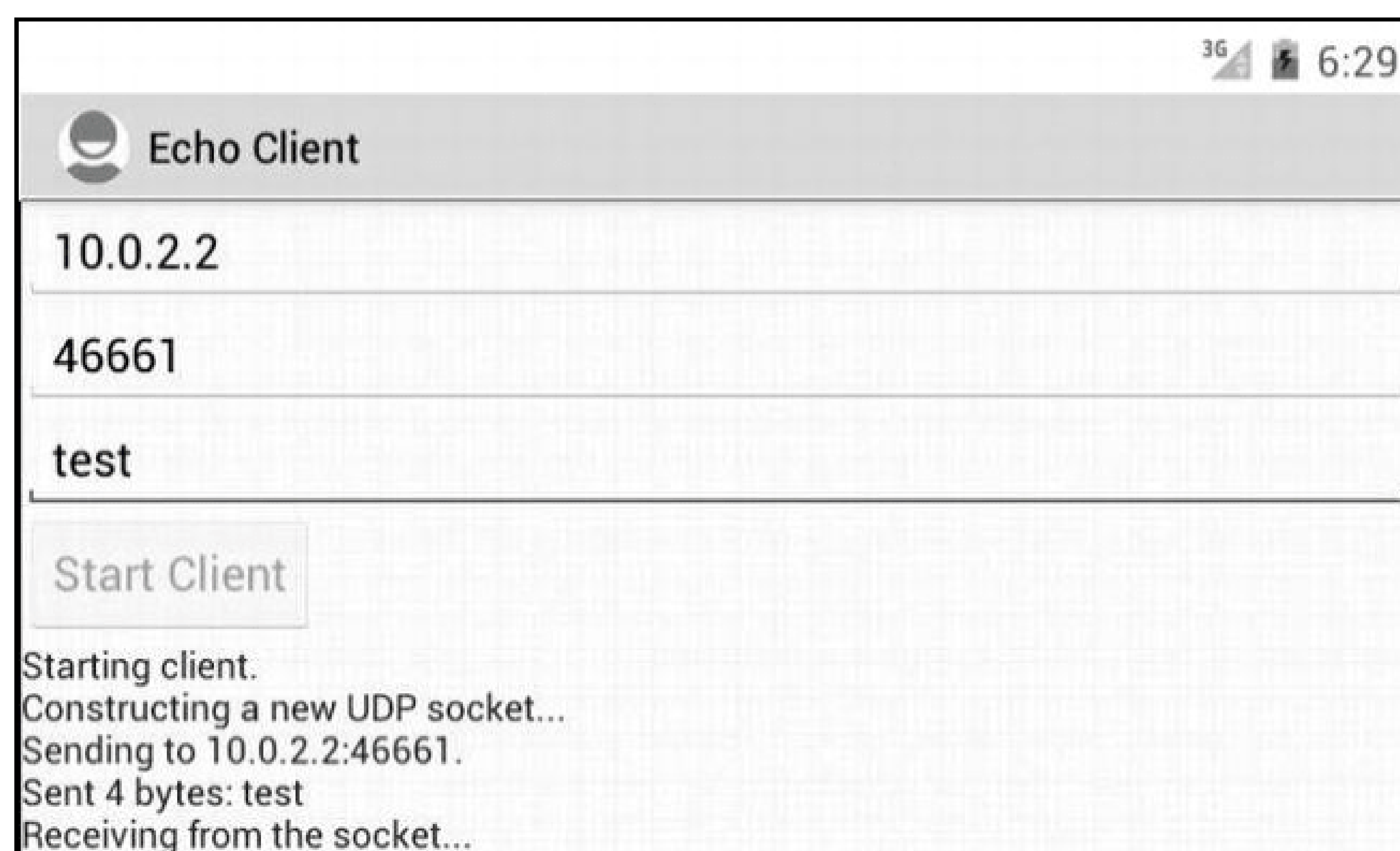


图 9-1　Echo UDP 客户端交换消息

9.6　小结

本章探讨了无连接通信的 POSIX Socket APIs ，学习了使用 UDP 协议的 client 和 server 模式，有了本章以及第 8 章所展示的核心概念做基础，可以虚拟地实现网络上提供的各种服务原生空间通信协议。第 10 章将演示如何用 POSIX Socket API 在设备的两个应用程序之间建立通信通道。

第10章

POSIX Socket API：本地通信

前两章我们学习了使用 POSIX Socket APIs 实现远程通信。POSIX Socket APIs 也可用于在同一设备上的两个应用程序之间或者原生代码与 Java 层之间建立本地通信通道。本章将继续在示例应用程序——Echo 上构建。本地 Socket 通信示例将讲解以下内容：

- 在原生层实现本地 socket 服务器
- 在 Java 层实现本地客户端
- 在两个应用程序之间建立本地 socket 通信

10.1 Echo Local Activity 布局

在 Project Explorer 视图中，展开 resources 下的 res 目录。展开 layout 子目录，创建一个名为 activity_echo_local.xml 的新布局文件。在 Editor 视图中，用程序清单 10-1 所示内容替换文件原来的内容。

程序清单 10-1 res/layout/activity_echo_local.xml 文件的内容

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <EditText
        android:id="@+id/port_edit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/local_port_edit" >

        <requestFocus />
```

```
        </EditText>

        <EditText
            android:id="@+id/message_edit"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="@string/message_edit" />

        <Button
            android:id="@+id/start_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/start_client_button" />

        <ScrollView
            android:id="@+id/log_scroll"
            android:layout_width="match_parent"
            android:layout_height="0dip"
            android:layout_weight="1.0" >

            <TextView
                android:id="@+id/log_view"
                android:layout_width="match_parent"
                android:layout_height="wrap_content" />
        </ScrollView>

    </LinearLayout>
```

Echo Local 提供了一个简单的用户界面以获取绑定本地 socket 的端口名、要发送的消息和运行时在原生本地 socket 服务器和客户端上显示状态更新信息。

10.2 Echo Local Activity

如前所述，打开 Project Explorer 视图，在 src 目录下新建一个名为 LocalSocketActivity.java 的类文件。在 Editor 视图中，添加程序清单 10-2 所示的内容。

程序清单 10-2 LocalSocketActivity.java 文件

```
package com.apress.echo;

import java.io.File;
import java.io.InputStream;
import java.io.OutputStream;
import java.nio.charset.Charset;

import android.net.LocalSocket;
import android.net.LocalSocketAddress;
```

```
import android.os.Bundle;
import android.widget.EditText;

/**
 * Echo 本地 socket 服务器和客户端.
 *
 * @author Onur Cinar
 */
public class LocalEchoActivity extends AbstractEchoActivity {
    /** 消息编辑. */
    private EditText messageEdit;

    /**
    * 构造函数.
    */
    public LocalEchoActivity() {
        super(R.layout.activity_local_echo);
    }

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        messageEdit = (EditText) findViewById(R.id.message_edit);
    }

    protected void onStartButtonClicked() {
        String name = portEdit.getText().toString();
        String message = messageEdit.getText().toString();

        if ((name.length() > 0) && (message.length() > 0)) {
            String socketName;

            // 如果是 filesystem socket，预先准备应用程序的文件目录
            if (isFilesystemSocket(name)) {
                File file = new File(getFilesDir(), name);
                socketName = file.getAbsolutePath();
            } else {
                socketName = name;
            }
            ServerTask serverTask = new ServerTask(socketName);
            serverTask.start();

            ClientTask clientTask = new ClientTask(socketName, message);
            clientTask.start();
        }
    }

    /**
     * 检查名称是否是 filesystem socket.
     *
```

```
     * @param name
     * socket 名称.
     * @return filesystem socket.
     */
    private boolean isFilesystemSocket(String name) {
        return name.startsWith("/");
    }

    /**
     * 启动绑定到给定名称的本地 UNIX socket 服务器。
     *
     * @param name
     * socket 名称.
     * @throws Exception
     */
    private native void nativeStartLocalServer(String name)
            throws Exception;

    /**
     * 启动本地 UNIX socket 客户端。
     *
     * @param port
     * 端口号.
     * @param message
     * 消息文本.
     * @throws Exception
     */
    private void startLocalClient(String name, String message)
        throws Exception {
        // 构造一个本地 socket
        LocalSocket clientSocket = new LocalSocket();
        try {
            // 设置 socket 名称空间
            LocalSocketAddress.Namespace namespace;
            if (isFilesystemSocket(name)) {
                namespace = LocalSocketAddress.Namespace.FILESYSTEM;
            } else {
                namespace = LocalSocketAddress.Namespace.ABSTRACT;
            }
            // 构造本地 socket 地址
            LocalSocketAddress address = new LocalSocketAddress(
                    name, namespace);

            // 连接到本地 socket
            logMessage("Connecting to " + name);
            clientSocket.connect(address);
            logMessage("Connected.");
            // 以字节形式获取消息
            byte[] messageBytes = message.getBytes();
```

```
            // 发送消息字节到 socket
            logMessage("Sending to the socket...");
            OutputStream outputStream = clientSocket.getOutputStream();
            outputStream.write(messageBytes);
            logMessage(String.format("Sent %d bytes: %s",
                    messageBytes.length, message));

            // 从 socket 中接收消息返回
            logMessage("Receiving from the socket...");
            InputStream inputStream = clientSocket.getInputStream();
            int readSize = inputStream.read(messageBytes);

            String receivedMessage = new String(messageBytes,
                    0, readSize);
            logMessage(String.format("Received %d bytes: %s",
                    readSize, receivedMessage));

            // 关闭流
            outputStream.close();
            inputStream.close();

        } finally {
            // 关闭本地 socket
            clientSocket.close();
        }
    }

    /**
     * 服务器任务.
     */
    private class ServerTask extends AbstractEchoTask {
        /** Socket 名称. */
        private final String name;

        /**
         * 构造函数.
         *
         * @param name
         * socket 名称.
         */
        public ServerTask(String name) {
            this.name = name;
        }

        protected void onBackground() {
            logMessage("Starting server.");

            try {
                nativeStartLocalServer(name);
            } catch (Exception e) {
```

```
                logMessage(e.getMessage());
            }

            logMessage("Server terminated.");
        }
    }

    /**
     * 客户端任务.
     */
    private class ClientTask extends Thread {
        /** Socket 名称. */
        private final String name;

        /** 发送的消息文本. */
        private final String message;

        /**
         * 构造函数.
         *
         * @parma name socket name.
         * @param message
         * 发送的消息文本.
         */
        public ClientTask(String name, String message) {
            this.name = name;
            this.message = message;
        }

        public void run() {
            logMessage("Starting client.");

            try {
                startLocalClient(name, message);
            } catch (Exception e) {
                logMessage(e.getMessage());
            }

            logMessage("Client terminated.");
        }
    }
}
```

LocalEchoActivity activity 获取了本地 socket 端口、来自 UI 的测试消息并创建了两个背景任务。第一个任务运行创建本地服务器 socket 并等待连接的本地 nativeStartLocalServer 方法，第二个任务运行用基于 Java 的 socket API 创建的、用于实现本地 socket 客户端与本地 socket 服务器通信的 startLocalClient Java 方法。与其他示例一样，与服务器 socket 连接后，客户端发送测试消息并等待服务器回显测试消息。

10.3　实现原生本地 Socket Server

在 Project Explorer 中，选择 LocalSocketActivty，然后在 External Tools 菜单中选择 Generate C and C++ Header File 以生成原生头文件。打开 Project Explorer，展开 jni 子目录，然后在编辑器中打开 Echo.cpp 源文件，将程序清单 10-3 所示内容插入文件头。

程序清单 10-3　包含 LocalSocketActivity 头文件

```
#include "com_apress_echo_LocalEchoActivity.h"
```

头文件包含 nativeStartLocalServer 原生方法声明。为了便于该原生方法的实现，需要先实现辅助函数集合。

10.3.1　创建本地 Socket：socket

用同一个 socket 函数创建本地 socket，这通过调用创建 PF_LOCAL 协议族中的 socket 的函数来实现。为了能够同时用所有类型的连接进行实验，不要修改已有的原生辅助函数，相反，而是要定义一个新的原生函数以创建本地 socket。在 Editor 视图中，将 NewLocalSocket 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 10-4 所示。

程序清单 10-4　NewLocalSocket 原生辅助函数

```
/**
 * 构造一个新的原生 UNIX socket.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @return socket descriptor.
 * @throws IOException
 */
static int NewLocalSocket(JNIEnv* env, jobject obj)
{
    // 构造 socket
    LogMessage(env, obj, "Constructing a new Local UNIX socket...");
    int localSocket = socket(PF_LOCAL, SOCK_STREAM, 0);
    // 检查 socket 构造是否正确
    if (-1 == localSocket)
    {
        // 抛出带错误号的异常
        ThrowErrnoException(env, "java/io/IOException", errno);
    }

    return localSocket;
}
```

本地 socket 族既支持基于流的 socket 协议，也支持基于数据报的 socket 协议。本例使用基于流的协议。

10.3.2 将本地 socket 与 Name 绑定：bind

与 TCP 及 UDP sockets 相同，一旦创建就不再需要分配协议地址，本地 socket 就在其 socket 族空间中存在。可以用同一个 bind 函数将本地 socket 与客户端用来连接的本地 socket 名绑定，通过 sockaddr_un 结构指定本地 socket 的协议地址，如程序清单 10-5 所示。

程序清单 10-5　sockaddr_un 地址结构体

```
struct sockaddr_un {
    sa_family_t sun_family;
    char sun_path[UNIX_PATH_MAX];
};
```

local socket 协议地址只由一个名字构成。它没有 IP 地址或者端口号，可以在两个不同的命名空间中创建本地 socket 名。

- Abstract namespace：在本地 socket 通信协议模块中维护，socket 名以 NULL 字符为前缀以绑定 socket 名。
- Filesystem namespace：通过文件系统以一个特殊 socket 文件的形式维护，socket 名直接传递给 sockaddr_un 结构，将 socket 名与 socket 绑定。

在 Editor 视图中，将 BindLocalSocketToName 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 10-6 所示。

程序清单 10-6　BindLocalSocketToName 原生辅助函数

```
/**
 * 将本地 UNIX socket 与某一名称绑定
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param sd socket descriptor.
 * @param name socket name.
 * @throws IOException
 */
static void BindLocalSocketToName(
        JNIEnv* env,
        jobject obj,
        int sd,
        const char* name)
{
    struct sockaddr_un address;

    // 名字长度
    const size_t nameLength = strlen(name);

    // 路径长度初始化与名称长度相等
    size_t pathLength = nameLength;

    // 如果名字不是以'/'开头，即它在抽象命名空间里
```

```
// in the abstract namespace
bool abstractNamespace = ('/' != name[0]);

// 抽象命名空间要求目录的第一个字节是 0 字节，更新路径长度包括 0 字节
if (abstractNamespace)
{
    pathLength++;
}

// 检查路径长度
if (pathLength > sizeof(address.sun_path))
{
    // 抛出带错误号的异常
    ThrowException(env, "java/io/IOException", "Name is too big.");
}
else
{
    // 清除地址字节
    memset(&address, 0, sizeof(address));
    address.sun_family = PF_LOCAL;

    // Socket 路径
    char* sunPath = address.sun_path;

    // 第一字节必须是 0 以使用抽象命名空间
    if (abstractNamespace)
    {
        *sunPath++ = NULL;
    }

    // 追加本地名字
    strcpy(sunPath, name);

    // 地址长度
    socklen_t addressLength =
            (offsetof(struct sockaddr_un, sun_path))
            + pathLength;

    // 如果 socket 名已经绑定，取消连接
    unlink(address.sun_path);

    // 绑定 socket
    LogMessage(env, obj, "Binding to local name %s%s.",
            (abstractNamespace) ? "(null)" : "",
            name);

    if (-1 == bind(sd, (struct sockaddr*) &address, addressLength))
    {
        // 抛出带错误号的异常
        ThrowErrnoException(env, "java/io/IOException", errno);
```

```
            }
        }
    }
```

BindLocalSocketToName 原生函数将给定的本地 socket 与给定的本地 socket 名绑定。它检查本地 socket 名是否以斜线开头以确定所使用的命名空间是 abstract 命名空间还是文件系统命名空间。一旦本地 socket 与本地 socket 名绑定，应用程序可能开始等待并接收即将到来的本地连接。

10.3.3　接受本地 Socket：accept

用同一个接收函数接收本地 socket 的输入连接，唯一的区别是由接收函数返回的客户端协议地址是 socketaddr_un 类型的。在 Editor 视图中，将 AcceptOnLocalSocket 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 10-7 所示。

程序清单 10-7　AcceptOnLocalSocket 原生辅助函数

```
/**
 * 阻塞并等待给定 socket 上即将到来的客户端连接.
 *
 * @param env JNIEnv interface.
 * @param obj object instance.
 * @param sd socket descriptor.
 * @return client socket.
 * @throws IOException
 */
static int AcceptOnLocalSocket(
        JNIEnv* env,
        jobject obj,
        int sd)
{
    // 阻塞并等待即将到来的客户端连接并且接收它
    LogMessage(env, obj, "Waiting for a client connection...");
    int clientSocket = accept(sd, NULL, NULL);

    // 如果客户端 socket 无效
    if (-1 == clientSocket)
    {
        // 抛出带错误号的异常
        ThrowErrnoException(env, "java/io/IOException", errno);
    }

    return clientSocket;
}
```

10.3.4　原生本地 Socket Server

nativeStartLocalServer 原生方法与 nativeStartTcpServer 原生方法非常相似，唯一的区别

是 nativeStartLocalServer 使用本地 socket 而不是 TCP socket。在 Editor 视图中，将 nativeStart-LocalServer 辅助函数追加到 Echo.cpp 原生模块源文件中，如程序清单 10-8 所示。

程序清单 10-8　nativeStartLocalServer 原生方法

```
void Java_com_apress_echo_LocalEchoActivity_nativeStartLocalServer(
        JNIEnv* env,
        jobject obj,
        jstring name)
{
    // 构造一个新的本地 UNIX socket.
    int serverSocket = NewLocalSocket(env, obj);
    if (NULL == env->ExceptionOccurred())
    {
        // 以 C 字符串形式获取名称
        const char* nameText = env->GetStringUTFChars(name, NULL);
        if (NULL == nameText)
            goto exit;

        // 绑定 socket 到某一端口号
        BindLocalSocketToName(env, obj, serverSocket, nameText);

        // 释放 name 文本
        env->ReleaseStringUTFChars(name, nameText);

        // 如果绑定失败
        if (NULL != env->ExceptionOccurred())
            goto exit;

        //监听有 4 个挂起连接的带 backlog 的 socket
        ListenOnSocket(env, obj, serverSocket, 4);
        if (NULL != env->ExceptionOccurred())
            goto exit;

        // 接受 socket 的一个客户连接
        int clientSocket = AcceptOnLocalSocket(env, obj, serverSocket);
        if (NULL != env->ExceptionOccurred())
            goto exit;

        char buffer[MAX_BUFFER_SIZE];
        ssize_t recvSize;
        ssize_t sentSize;

        // 接收并发送回数据
        while (1)
        {
            // 从 socket 中接收
            recvSize = ReceiveFromSocket(env, obj, clientSocket,
                    buffer, MAX_BUFFER_SIZE);

            if ((0 == recvSize) || (NULL != env->ExceptionOccurred()))
```

```
                    break;

                // 发送给 socket
                sentSize = SendToSocket(env, obj, clientSocket,
                        buffer, (size_t) recvSize);

                if ((0 == sentSize) || (NULL != env->ExceptionOccurred()))
                    break;
            }

            //关闭客户端 socket
            close(clientSocket);
        }

    exit:
        if (serverSocket > 0)
        {
            close(serverSocket);
        }
    }
```

nativeStartLocalServer 原生方法需要使用之前定义的辅助函数，它创建一个本地 socket 并将其与给定的名字绑定，开始等待本地连接并简单地回显接收到的字节。本地 socket 通信应用程序的服务器和客户端部分现在都实现了。

10.4 将本地 Echo Activity 添加到 Manifest 中

Echo local activity 需要添加到 Android Manifest 文件中才能使用。在 Project Explorer 视图的编辑器中打开 AndroidManifest.xml，用程序清单 10-9 所示内容修改文件内容。

程序清单 10-9 本地 Echo Activity 添加到 AndroidManifest.xml 文件

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.apress.echo"
    android:versionCode="1"
    android:versionName="1.0" >

        <activity
            android:name=".LocalEchoActivity"
            android:label="@string/title_activity_local_echo" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

</manifest>
```

10.5　运行本地 Sockets 示例

因为本地 socket 实例的服务器部分和客户端部分属于同一个 activity，可以按照以下步骤在单个 Android 模拟器实例上测试。

(1) 在 Android 模拟器上启动本地 socket activity。

(2) 将 Socket Name 设置成/file 以在 filesystem 命名空间中创建本地 socket。

(3) 将 Message 设置成将要传送的文本。

(4) 单击 Start 按钮启动客户端和服务器。

显示的 socket 事件和消息如图 10-1 所示。

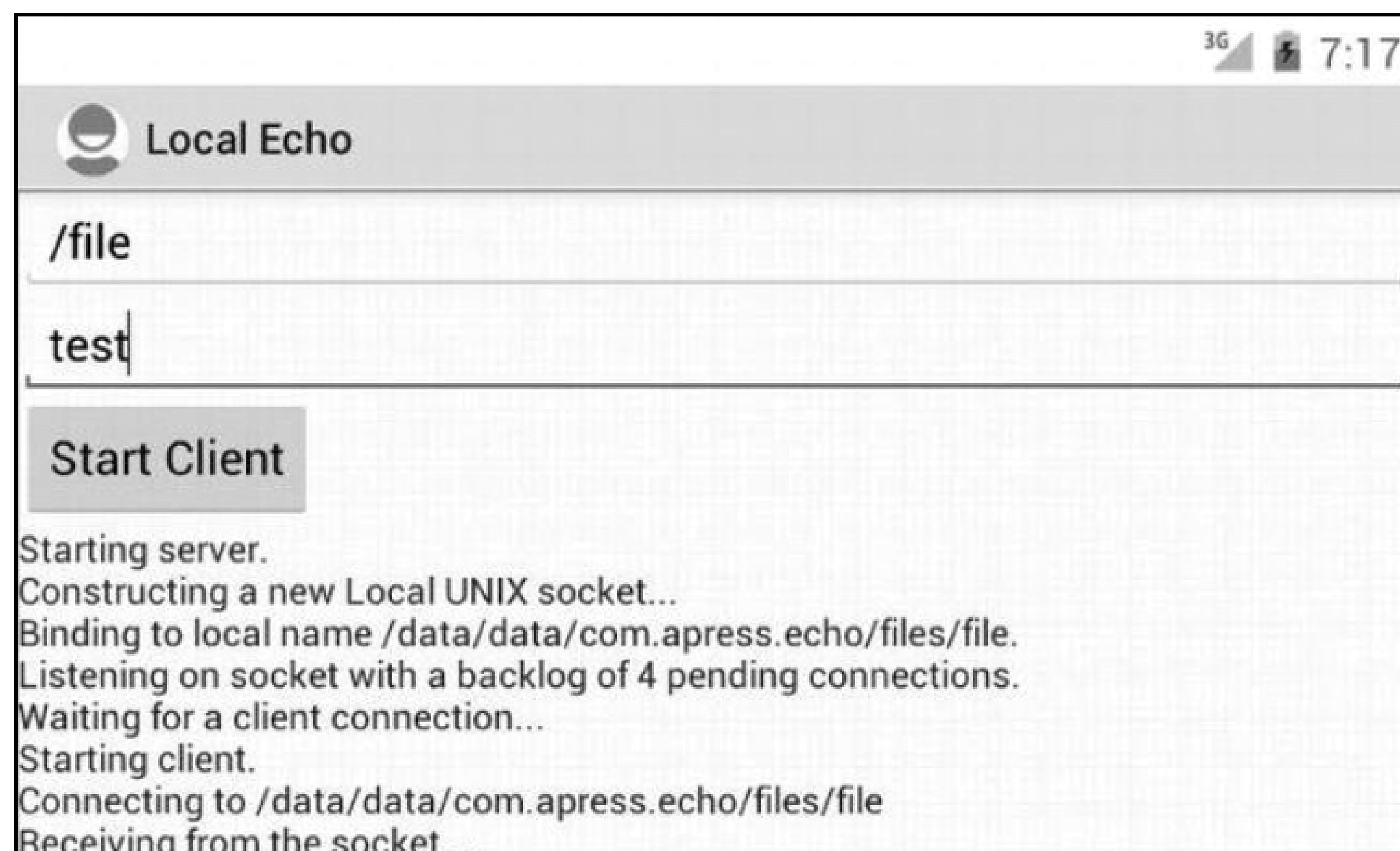


图 10-1　本地 echo 客户端和服务器端交换消息

10.6　异步 I/O

如前所述，大多数 socket APIs 阻塞函数调用。这些函数挂起调用进程直到满足某些条件，例如读操作时 socket 上有可读数据。socket 通过 select 函数提供异步 I/O。与其他在给定的时间内只能操作一个 socket 描述符的 socket APIs 不同，select 函数可以操作多个 socket 描述符并同时监控它们的状态。如果监控的一个事件发生或者到了指定的时限，则函数阻塞。要使用 select 函数，需要先包含 sys/select.h 头文件。

```
#include <sys/select.h>
```

select 函数要求提供下列参数：

```
int select(int nfds, fd_set* readfds, fd_set* writefds,fd_set* exceptfds,
struct timeval* timeout);
```

- nfds：为最高编号的描述符加 1，select 函数将监控 nfds 指定数量的描述符。
- readfds：设置将被监控可读性的描述符列表集。

- writefs：设置将被监控可写性的描述符列表集。
- exceptfds：设置将被监控任何类型错误的描述符列表集。
- timeout：指定为了完成选择而阻塞当前进程的最大时间间隔。如果不需要，将其值设置为 NULL。

如果选择成功，select 函数返回就绪的描述符数，否则返回-1，同时将 errno 设置为相应的错误。

select 函数的描述符列表通过 fd_set 结构提供。

```
struct fd_set readfds;
```

为了处理 manipulate 描述符列表，需要提供下面的宏集合：

- FD_ZERO 宏：保存指向 fd_set 结构的指针并清除它。
- FD_SET 宏：保存指向 fd_set 结构的指针并将描述符添加到集合中。
- FD_CLR 宏：保存指向 fd_set 结构的指针并从集合中删除描述符。
- FD_ISSET 宏：可以在选择完成后用来检查描述符是否为选择函数返回的集合的一部分。

10.7 小结

本章探讨了用于在同一台设备上的本地 socket 通信的 POSIX Socket APIs。本章还简要介绍了 POSIX Socket API 的异步 I/O 能力，后面三章(包括本章)将学习 Bionic 提供的、用于在原生层开发网络应用程序的基本概念和 API。学了这些内容，任何网络协议均可以很容易地在原生层实现。

第 11 章

支 持 C++

第 10 章已经探究了 Bionic C 标准库提供的功能，Bionic 提供了与操作系统以及硬件进行交互经常需要用到的一些基本结构和通用的抽象接口。与 Java 框架相比，Bionic 提供的通用结构确实很少。除了标准 C 库之外，C++ ISO 标准指定了针对 C++程序语言的额外标准库，即大家熟知的 C++标准库。该库提供了诸多泛型容器、字符串、流以及日常需要的一些实用工具函数。通过 C++标准库提供的搭建积木，通过使开发人员将精力集中在真正的程序逻辑上而不是开发一些实现逻辑所必须的结构，简化了原生开发过程。这使 C++开发效率更高，并促进了代码复用。

本章开始探究 Android 平台及 Android NDK 提供的 C++运行库支持。本章着重讲述以下核心内容：

- 各种可用的 C++运行库
- 异常以及 RTTI 支持的可用性
- C++标准库概念综述
- C++运行库线程安全
- C++运行库调试模式

11.1 支持的 C++运行库

Android 平台带有一个微型的 C++运行库支持库，称为系统运行库。该运行库不支持以下特性：

- C++标准库
- 异常支持
- RTTI 支持

Android NDK 提供了用于补充系统运行库功能的一些额外的 C++运行库，以完善上述缺失的特性。对这些 C++运行库的比较如表 11-1 所示。

表 11-1　支持的 C++运行库比较

C++运行库	C++异常支持	C++ RTTI 支持	C++标准库
系统库	No	No	No
GAbi++	No	Yes	No
STLport	No	Yes	Yes
GNU STL	Yes	Yes	Yes

11.1.1　GAbi++ C++运行库

GAbi++ C++运行库是一个实验性的、最简化型的运行库，它提供建立在系统运行库所提供的相同特性集基础之上的 RTTI 支持。它可以作为静态库或共享库使用。

11.1.2　STLport C++运行库

STLport 是一个开源的、多平台的 C++标准库实现。它提供一个 C++标准库头文件的完整集合以及对 RTTI 的支持。在本章编写时，Android NDK 中的 STLport C++运行库是基于 STLport 5.2.0 版本的。STLport 可以作为静态库或共享库使用。STLport 按免费许可证授权，可用于商业或开源项目中。

11.1.3　GNU STL C++运行库

GNU 标准 C++库，也叫 libstdc++-v3，是 Android NDK 中最全面的标准 C++运行库。它是一个正在开发的、以实现 ISO 标准 C++库为目标的开源项目。

在 GNU 标准 C++运行库中，C++异常与 C++RTTI 均被支持。如果原生代码确实需要任何一种特性，则需要通过构建系统变量进行显式的声明，正如在本章的 C++异常与 C++RTTI 部分中描述的那样。

在 Android NDK 中，GNU 标准 C++库可作为静态库或共享库使用。与 Android NDK 组件中的其他组件不同，除了 GCC Runtime Library Exception 之外，均是在 GNU 通用公共许可证版本 3(GNU GPL v3)许可下发布的。

11.2　指定 C++运行库

在原生 Android 项目中，Android NDK 构建系统变量 APP_STL 可被用于指定需要使用的 C++运行库。APP_STL 变量是一个应用级别的变量，仅可以在 jni 子目录下的 Application.mk 构建文件中定义，如程序清单 11-1 所示。

程序清单 11-1　选定 C++运行库的 jni/Application.mk 文件的内容

```
APP_ABI := armeabi armeabi-v7a
```

```
...
APP_STL := system
```

APP_STL 变量仅有一个值，即所用的 C++运行库的名字。在本章编写时，APP_STL 变量支持如下的值：

- system：默认的微型系统 C++运行库。若 APP_STL 未被设置，系统运行库会作为默认值被使用。
- gabi++_static：作为静态库的 GAbi++运行库。
- gabi++_shared：作为共享库的 GAbi++运行库。
- stlport_static：作为静态库的 STLport 运行库。
- stlport_shared：作为共享库的 STLport 运行库。
- gnustl_static：作为静态库的 GNU STL 运行库。
- gnustl_shared：作为共享库的 GNU STL 运行库。

11.3　静态运行库与动态运行库

除了系统运行库外，所有支持的 C++运行库都同时提供静态库和共享库。应用程序开发人员可选择将他们的原生模块与所需要的 C++运行库进行静态或动态的链接。

- 只有项目中包含单一的原生模块时支持静态库。
- 项目中包含多个原生模块时推荐使用共享库。

当 C++运行库以共享库的形式使用时，应用程序需要先加载所需要的共享库，然后再加载依赖于此共享库的其他原生模块。需要以逆序加载库文件，如程序清单 11-2 所示。

程序清单 11-2　显式加载动态 C++运行共享库

```
static {
    System.loadLibrary("strport_shared");
    System.loadLibrary("module1");
    System.loadLibrary("module2");
}
```

这将在加载原生模块前加载 stlport_shared 共享库，这样一来，在加载那些链接了 C++运行库的模块时，C++运行库处于可用状态。否则，对原生模块的加载将会失败。

11.4　C++异常支持

异常就是当出现一个异常事件时(如一段封装的代码块出错)，将程序的控制权转移给被称为异常处理程序的特定函数的机制。Android NDK 通过 GNU STL C++运行库提供对 C++异常的支持。为了在原生模块中使用 C++异常，需要按照如下方式在 Application.mk 中指定 GNU STL：

```
APP_STL := gnustl_shared
```

考虑到兼容性和性能的因素，默认情况下 C++ Exception 支持是不可用的。可以通过配置 Android.mk 构建文件中的 LOCAL_CPP_FEATURES 生成系统变量将 C++ Exception 支持设置为对单个原生模块可用。如程序清单 11-3 所示：

程序清单 11-3　Android.mk 构建文件启用 C++异常支持的配置内容

```
LOCAL_MODULE := module
...
LOCAL_CPP_FEATURES += exceptions
...
include $(BUILD_SHARED_LIBRARY)
```

可以通过配置 Application.mk 构建文件中的 APP_CPPFLAGS 生成系统变量将 C++异常支持授权给所有的原生模块，如程序清单 11-4 所示：

程序清单 11-4　Application.mk 生成文件启用 C++ Exceptions 支持的配置内容

```
APP_STL := gnustl_shared
APP_CPPFLAGS += -fexceptions
```

将 C++异常支持授权给所有的原生模块，这些模块只是应用程序的一部分。可用同样的方式启用 C++ RTTI 支持。

11.5　C++ RTTI 支持

运行库类型信息(Run-Time Type Information，RTTI)机制即在运行库展示对象类型信息。该机制主要用于执行安全类型转化。dynamic_cast、typeid 操作符还有 type_info 类是 RTTI 的一部分。Android NDK 通过 GAbi++、STLport 或者 GNU STL C++运行库给 RTTI 提供支持。为了在原生模块中使用 RTTI，首先需要按照如下方式在 Application.mk 中指定合适的 C++运行库。

```
APP_STL := gnustl_shared
```

考虑到兼容性和性能，默认情况下 C++ RTTI 支持是不可用的。可以通过配置 Android.mk 生成文件中的 LOCAL_CPP_FEATURES 构建系统变量将 C++异常支持授权给单个原生模块。如程序清单 11-5 所示：

程序清单 11-5　Android.mk 生成文件启用 RTTI 支持的配置内容

```
LOCAL_MODULE := module
...
LOCAL_CPP_FEATURES += rtti
...
include $(BUILD_SHARED_LIBRARY)
```

可以通过设置 Application.mk 生成文件中的 APP_CPPFLAGS 生成系统变量将 C++异常支持启用给所有的原生模块，如程序清单 11-6 所示：

程序清单 11-6　Application.mk 生成文件启用 RTTI 支持的配置内容

```
APP_STL := gnustl_shared
APP_CPPFLAGS += -frtti
```

将 C++ RTTI 支持授权给所有的原生模块，这些模块是应用程序的一部分。

11.6　C++标准库入门

因为 C++标准库规范非常庞大，本节只对其提供的功能做一个简要的概述。更多的信息可以在各自的 C++运行库文档中找到：

- STLport 文档的地址为：www.stlport.org/doc/
- GNU STL 文档的地址为：http://gcc.gnu.org/onlinedocs/libstdc++/

11.6.1　容器

容器是一个对象，可以存放其他对象，并提供访问和操作这些元素的方法。容器内元素的生命周期不会超过它所隶属的容器的生命周期。

1. 序列

序列是一个种大小可变的容器，它的元素都是线型排序的。C++标准库提供支持以下序列容器：

- vector 支持随机访问元素。它支持在末尾位置以常量时间插入和删除元素，在其他位置以线性时间插入和删除元素。
- deque 同 vector 类似，但除具备 vector 的结构属性外，还支持在序列开始的位置以常量时间插入和删除元素。这使得 deque 被选作实现队列的基础。
- list 是一个双向链表。list 同时支持正向和反向遍历序列。
- slist 是一个单项链表。slist 只支持正向遍历序列。

2. 关联容器

关联容器是一种大小可变的容器，它支持通过键来高效地检索元素。关联容器内的每个元素都有一个对应的键。关联容器主要有两种类型：排序关联容器和哈希关联容器。

排序关联容器

排序关联容器按照区分大小写升序排序来存储键值。它可以保证大多数操作的复杂度绝不会超过对数阶。下面是 C++标准库支持的排序关联容器：

- set 是一个已排序的简单关联容器。它的所有元素都已排序，而且没有任何两个元素是相同的。
- map 是用唯一的键来保存关联数据的关联容器。它使用键值与元素关联，没有任何两个元素是相同的。

- multiset 是一个已排序的、简单的、多重的关联容器。它的所有元素都是已排序的，而且允许有重复的元素。
- multimap 是一个已排序的、多重键值对的容器。它使用键值与元素关联，对拥有相同键值的元素数量不做限制。

哈希关联容器

哈希关联容器是基于哈希表实现的，它对存储的元素不做排序。与排序关联容器相比，哈希关联容器的速度要快很多。它可以保证大多数操作的最坏时间复杂度就是容器大小的线性操作复杂度。哈希关联容器的这个优势使得它特别适合快速查询需要的元素。同排序关联容器相比，哈希关联容器不会对它所存储的键值和元素做任何排序。下面是 C++标准库支持的哈希关联容器：

- hashed_set 是一个简单的散列关联容器，它不允许存在重复的元素。最适合快速检查集合中的一个元素。
- hash_map 是一个散列对关联容器，它将键和元素关联，而且可以通过这些键对元素进行快速查找。不管是键还是元素都没按照任何规则进行排序。
- hash_multiset 是一个简单的散列多重关联容器，它允许容器中出现出重复元素。同其他散列关联容器一样，它可以通过键来查找元素。
- hash_multimap 是一个散列对多重关联容器。它将键和元素进行关联而且提供快速查找。它允许容器中出现多个元素对应同一个键的情况。

3. 适配器

容器适配器用于在已有的基本容器类型的基础上提供专门的容器类型。一般通过限制已有的容器集合的功能来实现专门类型的容器。下面的容器就是通过适配器实现的：

- stack 是一种后进先出(LIFO)的数据结构。它是通过适配器限制 deque 的功能，并在 deque 的一端实现的。
- queue 是一种先进先出(FIFO)的数据结构。它同样是通过适配器限制 deque 的功能，并在 deque 的一端实现的。

4. String

String 同样是一个容器类型，它被表示为一个字符序列。除序列通常使用的方法外，string 类追加了标准的字符串操作方法，比如字符串串联和搜索。通过这些类提供的方法，字符串值可以和普通 C 字符串进行互相转换。

11.6.2 迭代器

迭代器可以对指定范围内或一个容器内的对象进行迭代。它们是泛化的指针，但是它们被实现成为通用类的形式。迭代器是 C++标准库的关键组成部分，因为它们是容器之间的接口和算法。C++标准库基于访问权限的级别和要执行的操作类型，提供了 5 种基本的迭代器：

- Input iterator：用来读取它所引用元素的值。
- Output iterator：由于修改当前位置对象的值。
- Forward iterator：可以用于多种算法，因为它符合值的线性序列的常用概念，而且它并不规定输入或输出操作。
- Bidirectional iterator：可以用于向前或向后遍历给定范围的元素。
- Random access iterator：提供普通 C 指针算法的所有操作，它为以任意大小步幅遍历元素提供常量时间方法。

通过适配器可以提供这些迭代器的派生迭代器，比如 reverse iterator 和 front insert iterator。

11.6.3　算法

C++标准库同样提供了一套广泛的常用功能来操作一系列的元素，如集合。对给定范围的元素，算法提供功能对其搜索、替换、复制和提取边界。它们依靠迭代器作为接口来遍历容器。

11.7　C++运行库的线程安全

所有的 C++运行库实现都是线程安全的，也就是说对共享容器的同时读操作时安全的，但是，如果线程既需要对共享容器进行读操作，又需要进行写操作，则应用程序负责确保操作的互斥性。

11.8　C++运行库调试模式

C++运行库的性能已经进行优化，所以它们很少或根本不执行错误检查。GNU STL 和 STLport 的 C++运行库提供了调试模式，使得检测对 C++标准库错误的使用和隐藏在应用程序代码中的 bug 变得更容易。调试模式用语义上对等但安全的容器和迭代器来替换不安全的标准容器和迭代器。以下是调试模式提供的调试工具：

- Safe iterators：追踪和连接着迭代器的容器。它们会执行迭代器的有效性和所有权的运行库校验。比如，向一个指向一个已被销毁容器的迭代器取值，只要存在这样的错误，在调试模式下就会被发现。
- Algorithm：预处理尝试验证输入的参数。并且只要存在错误就会被检测出来。算法预处理会用任何有效的附加信息又来验证，如迭代器在容器中的位置。

11.8.1　GNU STL 调试模式

GNU STL C++运行库允许在指定的部分代码或整个应用程序下启动调试模式。

1. 使用单独的 GNU STL 调试容器

为了只为指定的部分代码启动调试模式，GNU STL 用__gnu_debug 命名空间代替 std 命名空间来为大多数的容器提供调试模式启动副本。这些调试容器可以包含在头文件名加有 debug 前缀的子目录下，如程序清单 11-7 所示：

程序清单 11-7　在部分代码内启动 GNU STL 调试模式

```
// 包含调试 vector 容器
#include <debug/vector>
...
__gnu_debug::vector v;
```

使用单独的 GNU STL 调试容器需要修改代码，在大部分情况下这不是最优的选择。GNU STL 调试模式同样可以在不需要修该源代码的情况下，在编译时启动。

2. 启动 GNU STL 调试模式

可以通过预处理标识_GLIBCXX_DEBUG 来控制调试模式。可以通过 APP_CFLAGS 构建系统变量为项目中的所有原生模块定义该标识，也可以通过 LOCAL_CFLAGS 构建系统变量为指定的原生模块定义该标识，如程序清单 11-8 所示。

程序清单 11-8　为当前模块启用 GNU STL 调试模式

```
LOCAL_MODULE := module
...
LOCAL_CFLAGS += -D_GLIBCXX_DEBUG
...
include $(BUILD_SHARED_LIBRARY)
```

原生模块会根据启用或者禁用调试模式的情况被重新编译。

11.8.2　STLport 调试模式

可以通过预处理标识_STLP_DEBUG 来控制调试模式。可以通过 APP_CFLAGS 构建系统变量为项目中的所有原生模块定义该标识，也可以通过 LOCAL_CFLAGS 构建系统变量为指定的原生模块定义该标识，如程序清单 11-9 所示。

程序清单 11-9　为当前模块启用 STLport 调试模式

```
LOCAL_MODULE := module
...
LOCAL_CFLAGS += -D_STLP_DEBUG
...
include $(BUILD_SHARED_LIBRARY)
```

原生模块需要根据启用或者禁用调试模式的情况被重新编译。

将调试模式信息重定向到 Android 日志

默认情况下，错误信息会显示在标准错误输出里。STLport 可以通过将错误信息重定向到自定义函数来重写默认的行为。要实现这一点，请按照下列步骤操作。

(1) 修改 Android.mk 构建文件定义_STLP_DEBUG_MESSAGE 预处理宏，如程序清单 11-10 所示：

程序清单 11-10　启用自定义调试信息输出函数

```
LOCAL_MODULE := module
...
LOCAL_CFLAGS += -D_STLP_DEBUG
LOCAL_CFLAGS += -D_STLP_DEBUG_MESSAGE
LOCAL_LDLIBS += -llog
...
include $(BUILD_SHARED_LIBRARY)
```

(2) 实现全局函数__stl_debug_message 将错误信息重定向到 Android 日志，如程序清单 11-11 所示。

程序清单 11-11　__stl_debug_message 函数的实现

```
#include <stdarg.h>
#include <android/log.h>
...
void __stl_debug_message(const char* format_str, ...)
{
    va_list ap;

    va_start(ap, format_str);
    __android_log_vprint(ANDROID_LOG_FATAL, "STLport", format_str, ap);
    va_end(ap);
}
```

根据上面作出的修改，STL 调试信息和 STLport 标签、日志级别 FATAL 会被重定向到 Android 日志，而且这些日志可以通过 logcat 进行监控。

11.9　小结

本章开始探讨通过 Android 平台和 Android NDK 提供的 C++运行库支持。根据所提供的功能对 Android NDK 支持的不同 C++运行库进行了比较，如 C++异常支持和 C++ RTTI 支持。C++标准库是非常庞大和复杂，本章介绍了线程安全和调试模式的 C++运行库，用来排除原生应用中对 C++ 组件无效使用的相关故障。第 12 章将看到 C++标准库函数在应用中的示例。

第12章

原生图形 API

毋庸置疑，游戏和多媒体应用程序是从 Android NDK 中获益最多的。这些应用程序依靠原生代码来进行性能相关的操作。对于这些应用程序来说，能从原生代码层直接向显示器渲染图像是至关重要的。本章将探讨下面几个 Android NDK 提供的原生图形 API：

- JNI 图形 API(也叫位图 API)
- OpenGL ES 1.x and 2.0
- 原生 Window API

我们将在本章用各种可用的原生图形 API 构建一个 AVI 视频播放应用程序，以此作为演示视频帧渲染的测试平台。

12.1　原生图形 API 的可用性

并不是所有的原生图形 API 对各个版本的 Android 操作系统都是可用的。随着时间的推移所介绍的这些 API 只适用于众多 Android 版本的一个子集。可用的原生图形 API 如表 12-1 所示：

表 12-1　可用的原生图形 API

Native Graphics API	Android Version	API Level
JNI Graphics API	2.2 及以后	8 及以后
OpenGL ES 1.x	1.6 及以后	4 及以后
OpenGL ES 2.0	2.0 及以后	5 及以后
原生 Window	2.3 及以后	9 及以后

在深入探究用原生代码显示图形之前，我们将先创建一个简单的 AVI 视频播放应用程序。

12.2 创建一个 AVI 视频播放器

这个 AVI 视频播放应用程序将充当一个测试平台。在本章中，我们将不断地扩展这个测试应用程序来试验在原生空间中各种不同的可用原生图形 API，该示例应用程序包括以下功能：

- 它是一个支持原生代码的 Android 应用程序项目。
- 一个静态链接 AVI 库，其中包含了一些提供给 Java 层使用的基本函数并与 activity 的生命周期绑定。
- 一个简单的 GUI 界面来指定播放的 AVI 视频文件的名字以及原生图形 API 的类型。

播放 AVI 视频文件需要解析 AVI 文件。虽然 AVI 文件格式不是非常复杂，但为了简单起见，我们将使用一个叫做 AVILib 的第三方 AVI 库来处理 AVI 文件。

12.2.1 将 AVILib 作为 NDK 的一个导入模块

AVILib 库来自于一个名为 Transcode 的大型开源项目。可以按照下面的步骤使 AVILib 成为一个 NDK 导入模块以便使用。

(1) 用你喜欢的浏览器打开 http://tcforge.berlios.de/。

(2) 编写本书时，Transcode 的最新版本是 1.1.5。单击 Download 链接下载 transcode-1.1.5.tar.bz2 源压缩包。

(3) 如果你用的是 Mac OS 或者 Linux，打开一个终端窗口；如果使用 Windows，打开 Cygwin。

(4) 在命令行输入下面的命令将 Android NDK 导入模块目录改为当前工作目录。

```
cd $ANDROID_NDK_HOME/sources
```

(5) transcode 的源压缩文件是用 BZip2 压缩的 TAR 文件。用下载的 transcode-1.1.5.tar.bz2 文件的存放目录名字替换掉<Download Location>，然后输入以下命令来提取压缩包中的文件：

```
tar jxvf <Download Location>/transcode-1.1.5.tar.bz2
```

(6) 用以下的命令将当前的目录改为 Transcode 的 avilib 子目录：

```
cd transcode-1.1.5/avilib
```

(7) 在 Eclipse 中打开 platform.h 头文件。在 config.h 头文件包含附近添加粗体行的语句，如程序清单 12-1 所示。

程序清单 12-1　修改 AVILIB platform.h 头文件的内容

```
#ifndef PLATFORM_H
#define PLATFORM_H

#ifdef HAVE_CONFIG_H
```

```
#include "config.h"
#endif

#ifdef OS_DARWIN
#include <sys/uio.h>
#endif
```

之所以这样修改是因为将通过 Android NDK 构建系统而不是 Transcode 项目的 Makefile 来完成 AVILIB 的编译。

(8) Android NDK 构建系统需要它自己的 Android.mk 文件中的导入模块。打开 Eclipse，在当前目录下创建一个新的 Android.mk，内容如程序清单 12-2 所示。

程序清单 12-2　用于 AVILib 导入模块的 Android.mk 构建文件

```
LOCAL_PATH := $(call my-dir)

#
# 转码 AVILib
#

# 源文件
MY_AVILIB_SRC_FILES := avilib.c platform_posix.c

# 包含导出路径
MY_AVILIB_C_INCLUDES := $(LOCAL_PATH)

#
# AVILib 静态
#
include $(CLEAR_VARS)

# 模块名称
LOCAL_MODULE := avilib_static

# 源文件
LOCAL_SRC_FILES := $(MY_AVILIB_SRC_FILES)

# 包含导出路径
LOCAL_EXPORT_C_INCLUDES := $(MY_AVILIB_C_INCLUDES)

# 构建静态库
include $(BUILD_STATIC_LIBRARY)

#
# AVILib 共享
#
include $(CLEAR_VARS)

# 模块名称
```

```
LOCAL_MODULE := avilib_shared

# 源文件
LOCAL_SRC_FILES := $(MY_AVILIB_SRC_FILES)

# 包含导出路径
LOCAL_EXPORT_C_INCLUDES := $(MY_AVILIB_C_INCLUDES)

# 构建共享库
include $(BUILD_SHARED_LIBRARY)
```

该构建脚本为 AVILib 库定义了一个静态导入模块和一个共享导入模块。

AVILib 库现在准备好了。下面将着手实现 AVI Player 示例项目，它将用 AVILib 库来播放 AVI 格式的视频文件。

12.2.2 创建 AVI 播放器 Android 应用程序

正如本书前面所述，在 Eclipse 中启动 New Android Application Project 对话框，完成以下步骤：

(1) 将 Application Name 设置为 AVI Player。

(2) 将 Project Name 设置为 AVI Player。

(3) 将 Package Name 设置为 com.apress.aviplayer。

(4) 单击 Next 按钮接受其他设置采用默认值。

(5) 单击 Next 按钮接受使用默认的启动图标。

(6) 取消选中 Create Activity，并单击 Finish 按钮创建一个空的 AVI Player 项目。

(7) 想要添加原生支持，打开 Project Explorer，通过 Android Tools 的上下文菜单打开 Add Android Native Support 向导。

(8) 将 Library Name 设置为 AVIPlayer。

(9) 单击 Finish 按钮将原生支持添加到 AVI Player 项目。

12.2.3 创建 AVI Player 的 Main Activity

main activity 将提供一个简单的 GUI 界面，用于指定 AVI 视频文件名和用于渲染的原生图形 API 类型。打开 Eclipse，选择顶部菜单栏的 New | Other，展开 Android，选择 Android Activity，并且单击 Next 启动一个新的 Android Activity 对话框，完成以下步骤：

(1) 选择 Blank Activity 模板。

(2) 单击 Next 按钮继续。

(3) 设置 Activity 的名字为 MainActivity。

(4) 单击 Finish 按钮接受默认的设置并创建一个新的 activity。

(5) 打开 Project Explorer，打开 AndroidManifest.xml 清单文件，用程序清单 12-3 的内容替换文件内容。

程序清单 12-3　AndroidManifest.xml 文件的内容

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.apress.aviplayer"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="15" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/main_activity_title" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
                      android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

(6) 打开 Project Explorer，展开 res 资源目录。进入到 values 子目录，打开 strings.xml 字符串资源文件。用程序清单 12-4 中的内容替换文件内容。

程序清单 12-4　res/values/strings.xml 资源文件的内容

```
<resources>
    <string name="app_name">AVI Player</string>
    <string name="main_activity_title">MainActivity</string>
    <string name="file_name_hint">AVI Video File Name</string>
    <string name="file_name_text">galleon.avi</string>
    <string name="play_button">Play</string>
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>
    <string name="error_alert_title">Error Occurred</string>
</resources>
```

(7) main activity 提供一个非常简单的 GUI，文本区域用来指定 AVI 文件名、单选按钮用来选择所使用的原生图形 API。打开 Project Explorer，展开 res 目录下的 layout 子目录。打开 activity_main.xml 布局文件并用程序清单 12-5 的内容替换文件内容。

程序清单 12-5　res/layout/activity_main.xml 布局文件的内容

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <EditText
        android:id="@+id/file_name_edit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="@string/file_name_hint"
        android:text="@string/file_name_text" >

        <requestFocus />
    </EditText>

    <RadioGroup
        android:id="@+id/player_radio_group"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >

    </RadioGroup>

    <Button
        android:id="@+id/play_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/play_button" />

</LinearLayout>
```

(8) 最后，要实现 activity。使用 Project Explorer 打开 MainActivity.java 源文件并用程序清单 12-6 的内容替换文件内容。

程序清单 12-6　MainActivity.java 源文件的内容

```
package com.apress.aviplayer;

import java.io.File;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
```

```
import android.widget.EditText;
import android.widget.RadioGroup;

/**
 * Main activity.
 *
 * @author Onur Cinar
 */
public class MainActivity extends Activity implements OnClickListener {
    /** AVI 文件名字编辑*/
    private EditText fileNameEdit;

    /** Player 类型的单选组*/
    private RadioGroup playerRadioGroup;

    /** Play 按钮*/
    private Button playButton;

    /**
     * On create.
     *
     * @param savedInstanceState saved state.
     */
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        fileNameEdit = (EditText) findViewById(R.id.file_name_edit);
        playerRadioGroup = (RadioGroup) findViewById(
                R.id.player_radio_group);

        playButton = (Button) findViewById(R.id.play_button);
        playButton.setOnClickListener(this);
    }

    /**
     *OnClick 事件处理
     *
     * @param view view instance.
     */
    public void onClick(View view) {
        switch (view.getId()) {
        case R.id.play_button:
            onPlayButtonClick();
            break;
        }
    }

    /**
     *按钮单击时的事件处理
```

```
     */
    private void onPlayButtonClick() {
        Intent intent;

        //获得选择的单选按钮的 id
        int radioId = playerRadioGroup.getCheckedRadioButtonId();

        // 基于 id 选择 activity
        switch (radioId) {

        //本章的后面会在此处添加 case 代码
        default:
            throw new UnsupportedOperationException(
                    "radioId=" + radioId);
        }

        // 基于外部存储器
        File file = new File(Environment.getExternalStorageDirectory(),
                fileNameEdit.getText().toString());

        //将 AVI 文件的名字作为 extra 内容
        intent.putExtra(AbstractPlayerActivity.EXTRA_FILE_NAME,
                file.getAbsolutePath());

        // 启动 player activity
        startActivity(intent);
    }
}
```

12.2.4 创建 Abstract Player Activity

当用不同的原生图形 API 做实验时，AVI 播放器代码的实现很大程度上是相同的，例如打开或者关闭 AVI 文件。抽象 player activity 会提取公共的代码，只让子类通过继承它来完成实际播放器实现的渲染部分。按照以下步骤来实现 abstract player activity。

(1) 打开 Project Explorer，展开 src 目录。

(2) 右击 com.apress.aviplayer 包。

(3) 在上下文菜单中选择 New | Class 启动一个 New Java Class 对话框。

(4) 将 Name 设置为 AbstractPlayerActivity。

(5) 单击 Finish 按钮来创建一个新的类。

(6) 用程序清单 12-7 中的内容替换 AbstractPlayerActivity.java 源文件的内容。

程序清单 12-7 AbstractPlayerActivity.java 源文件的内容

```
package com.apress.aviplayer;

import java.io.IOException;

import android.app.Activity;
```

```
import android.app.AlertDialog;

/**
 * Player activity.
 *
 * @author Onur Cinar
 */
public abstract class AbstractPlayerActivity extends Activity {
    /** AVI 文件名字的 extra */
    public static final String EXTRA_FILE_NAME =
            "com.apress.aviplayer.EXTRA_FILE_NAME";

    /** AVI 视频文件描述符*/
    protected long avi = 0;

    /**
     * On start.
     */
    protected void onStart() {
        super.onStart();

        // 打开 AVI 文件
        try {
            avi = open(getFileName());
        } catch (IOException e) {
            new AlertDialog.Builder(this)
                    .setTitle(R.string.error_alert_title)
                    .setMessage(e.getMessage())
                    .show();
        }
    }

    /**
     * On stop.
     */
    protected void onStop() {
        super.onStop();

        //如果 AVI 视频是打开的
        if (0 != avi) {
            //关闭文件描述符
            close(avi);
            avi = 0;
        }
    }

    /**
     *获取 AVI 视频文件的名字
     *
     * 返回文件名字
```

```
     */
    protected String getFileName() {
        return getIntent().getExtras().getString(EXTRA_FILE_NAME);
    }

    /**
     *打开指定的 AVI 文件并且返回一个文件描述符
     *
     * @param fileName file name.
     * @return file descriptor.
     * @throws IOException
     */
    protected native static long open(String fileName)
            throws IOException;

    /**
     *获得视频宽度
     *
     * @param avi file descriptor.
     * @return video width.
     */
    protected native static int getWidth(long avi);

    /**
     *获得视频高度
     *
     * @param avi file descriptor.
     * @return video height.
     */
    protected native static int getHeight(long avi);

    /**
     *获得帧速
     *
     * @param avi file descriptor.
     * @return frame rate.
     */
    protected native static double getFrameRate(long avi);

    /**
     *基于给定的文件描述符关闭指定的 AVI 文件
     *
     * @param avi file descriptor.
     */
    protected native static void close(long avi);

    static {
        System.loadLibrary("AVIPlayer");
    }
}
```

AbstractPlayerActivity 还包含一些用来处理 AVI 视频文件的原生方法。这些方法需要在原生空间中实现。

(7) 在顶部菜单栏中选择 Project | Build Project 来编译 Java 源代码。之后可以使用 javah 工具生成为实现 AbstractPlayerActivity 原生部分的必要的头文件。

(8) 打开 Project Explorer，选中 AbstractPlayerActivity。

(9) 在顶部菜单栏中选择 Run | External Tools | Generate C and C++ Header File 来为 AbstractPlayerActivity 类调用 javah 工具。

(10) 在项目的 jni 子目录下，javah 工具将生成 com_apress_aviplayer_AbstractPlayerActivity.h 头文件，内容如程序清单 12-8 所示。

程序清单 12-8　com_apress_aviplayer_AbstractPlayerActivity.h 的内容

```
/*不要编辑这个文件，它是由系统生成的*/
#include <jni.h>
/*类 com_apress_aviplayer_AbstractPlayerActivity 的头文件 */

#ifndef _Included_com_apress_aviplayer_AbstractPlayerActivity
#define _Included_com_apress_aviplayer_AbstractPlayerActivity
#ifdef __cplusplus
extern "C" {
#endif

...

/*
 * Class: com_apress_aviplayer_AbstractPlayerActivity
 * Method: open
 * Signature: (Ljava/lang/String;)J
 */
JNIEXPORT jlong JNICALL Java_com_apress_aviplayer_AbstractPlayerActivity_open
  (JNIEnv *, jclass, jstring);

/*
 * Class: com_apress_aviplayer_AbstractPlayerActivity
 * Method: getWidth
 * Signature: (J)I
 */
JNIEXPORT jint JNICALL Java_com_apress_aviplayer_AbstractPlayerActivity_
getWidth
  (JNIEnv *, jclass, jlong);

/*
 * Class: com_apress_aviplayer_AbstractPlayerActivity
 * Method: getHeight
 * Signature: (J)I
 */
JNIEXPORT jint JNICALL Java_com_apress_aviplayer_AbstractPlayerActivity_
getHeight
```

```
  (JNIEnv *, jclass, jlong);

/*
 * Class: com_apress_aviplayer_AbstractPlayerActivity
 * Method: getFrameRate
 * Signature: (J)D
 */
JNIEXPORT jdouble JNICALL Java_com_apress_aviplayer_AbstractPlayerActivity_
getFrameRate
  (JNIEnv *, jclass, jlong);

/*
 * Class: com_apress_aviplayer_AbstractPlayerActivity
 * Method: close
 * Signature: (J)V
 */
JNIEXPORT void JNICALL Java_com_apress_aviplayer_AbstractPlayerActivity_
close
  (JNIEnv *, jclass, jlong);

#ifdef __cplusplus
}
#endif
#endif
```

(11) 为了实现这些原生函数，需要一个新的 C++源文件。右击 jni 目录，在上下文菜单中选择 New | Source File。

(12) 将 Source File 设置为 com_apress_aviplayer_AbstractPlayerActivity.cpp。

(13) 单击 Finish 按钮来创建一个新的 C++源文件。

(14) Abstract player activity 的原生方法提供了通过 AVILib 这个第三方库所提供的 API 解析指定 AVI 视频文件的功能。打开 Eclipse，用程序清单 12-9 的内容替换 com_apress_aviplayer_AbstractPlayerActivity.cpp 源文件的内容。

程序清单 12-9　com_apress_aviplayer_AbstractPlayerActivity.cpp 的内容

```
extern "C" {
#include <avilib.h>
}

#include "Common.h"
#include "com_apress_aviplayer_AbstractPlayerActivity.h"

jlong Java_com_apress_aviplayer_AbstractPlayerActivity_open(
        JNIEnv* env,
        jclass clazz,
        jstring fileName)
{
    avi_t* avi = 0;
```

```
    //获取文件名字赋给 C 的一个字符串变量
    const char* cFileName = env->GetStringUTFChars(fileName, 0);
    if (0 == cFileName)
    {
        goto exit;
    }

    //打开 AVI 文件
    avi = AVI_open_input_file(cFileName, 1);

    // 释放文件名字
    env->ReleaseStringUTFChars(fileName, cFileName);

    //如果 AVI 文件不能打开则抛出一个异常
    if (0 == avi)
    {
        ThrowException(env, "java/io/IOException", AVI_strerror());
    }

exit:
    return (jlong) avi;
}

jint Java_com_apress_aviplayer_AbstractPlayerActivity_getWidth(
        JNIEnv* env,
        jclass clazz,
        jlong avi)
{
    return AVI_video_width((avi_t*) avi);
}

jint Java_com_apress_aviplayer_AbstractPlayerActivity_getHeight(
        JNIEnv* env,
        jclass clazz,
        jlong avi)
{
    return AVI_video_height((avi_t*) avi);
}

jdouble Java_com_apress_aviplayer_AbstractPlayerActivity_getFrameRate(
    JNIEnv* env,
    jclass clazz,
    jlong avi)
{
    return AVI_frame_rate((avi_t*) avi);
}

void Java_com_apress_aviplayer_AbstractPlayerActivity_close(
        JNIEnv* env,
        jclass clazz,
```

```
        jlong avi)
{
    AVI_close((avi_t*) avi);
}
```

(15) 不同播放器实现的 Abstract player activity 原生代码部分将会共享一些通用代码。这些通用代码将由 Common.h 和 Common.cpp 源文件提供。右击 jni 目录，在上下文菜单中选择 New | Header File。

(16) 将 Header File 设置为 Common.h。

(17) 单击 Finish 按钮创建一个新的头文件。

(18) 用程序清单 12-10 的内容替换新头文件的内容。

程序清单 12-10　Common.h 头文件的内容

```
#pragma once

#include <jni.h>

/**
 *使用给定的异常类和异常信息抛出一个新的异常
 * *
 * @param env JNIEnv interface.
 * @param className class name.
 * @param message exception message.
 */
void ThrowException(
        JNIEnv* env,
        const char* className,
        const char* message);
```

(19) 右击 jni 目录，在上下文菜单中选择 New | Source File。

(20) 将 Source File 设置为 Common.cpp。

(21) 单击 Finish 按钮来创建一个新的 C++源文件。

(22) 用程序清单 12-11 的代码替换新源文件的内容。

程序清单 12-11　Common.cpp 源文件的内容

```
#include "Common.h"

void ThrowException(
        JNIEnv* env,
        const char* className,
        const char* message)
{
    // 获取异常类
    jclass clazz = env->FindClass(className);

    // 如果找到异常类
    if (0 != clazz)
```

```
        {
            // 抛出异常
            env->ThrowNew(clazz, message);

            // 释放本地类引用
            env->DeleteLocalRef(clazz);
        }
    }
```

(23) 现在应该更新原生项目的构建文件使其包含新的源文件，同时静态地链接到 AVILib 第三方模块上。在 jni 子目录中打开 Android.mk 文件并用程序清单 12-12 的代码替换文件内容。

程序清单 12-12　Android.mk 构建文件的内容

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := AVIPlayer
LOCAL_SRC_FILES := \
    Common.cpp \
    com_apress_aviplayer_AbstractPlayerActivity.cpp
# 使用 AVILib 静态库
LOCAL_STATIC_LIBRARIES += avilib_static

include $(BUILD_SHARED_LIBRARY)

# 引入 AVILib 库模块
$(call import-module, transcode-1.1.5/avilib)
```

(24) 虽然现在还没实现 AVI 视频播放的渲染功能，但也要构建它并在模拟器上运行该示例应用程序，以确保在进行下一步之前它已被正确实现。

12.3　使用 JNI 图形 API 进行渲染

Android 框架提供了 android.graphics.Bitmap 类用来在 Java 代码中操作和使用 bitmap 像素缓存。从 Android 2.2(API Level 8)开始，Android 提供了 JNI Graphics API，可以使用原生代码访问和操作 Bitmap 对象的像素缓存。

12.3.1　启用 JNI Graphics API

在你的原生应用程序中，按照以下步骤进行操作以使用 JNI Graphics API：

(1) 包含 android/bitmap.h 头文件。

```
#include <android/bitmap.h>
```

(2) 修改 Android.mk 构建文件以动态地与 jnigraphics 库进行链接。

```
LOCAL_LDLIBS += -ljnigraphics
```

做了以上修改之后，现在 JNI 图形 API 已经可以你的原生应用程序中使用了。

12.3.2 使用 JNI Graphics API

JNI Graphics API 提供了 4 个原生函数用于访问和操作 Bitmap 对象。

1. 检索 Bitmap 对象信息

AndroidBitmap_getInfo 函数允许原生代码检索 Bitmap 对象信息，如它的大小、像素格式等。

```
int AndroidBitmap_getInfo(JNIEnv* env,
                          jobject bitmap,
                          AndroidBitmapInfo* info);
```

该函数以 JNI 接口指针、Bitmap 对象的引用、一个指向 AndroidBitmapInfo 结构体的指针为参数，该结构返回指定 bitmap 的信息，如程序清单 12-13 所示。

程序清单 12-13 AndroidBitmapInfo 结构体的声明

```
typedef struct {
    uint32_t width;
    uint32_t height;
    uint32_t stride;
    int32_t format;
    uint32_t flags;
} AndroidBitmapInfo;
```

format 字段包含了像素格式信息，如程序清单 12-14 所示。

程序清单 12-14 AndroidBitmapFormat 枚举的声明

```
enum AndroidBitmapFormat {
    ANDROID_BITMAP_FORMAT_NONE      = 0,
    ANDROID_BITMAP_FORMAT_RGBA_8888 = 1,
    ANDROID_BITMAP_FORMAT_RGB_565   = 4,
    ANDROID_BITMAP_FORMAT_RGBA_4444 = 7,
    ANDROID_BITMAP_FORMAT_A_8       = 8,
};
```

成功的话，AndroidBitmap_getInfo 函数会返回 0；否则会返回一个负数。在 android/bitmap.h 头文件中可以找到全部的错误码列表。

2. 访问原生像素缓存

AndroidBitmap_lockPixels 函数锁定了像素缓存以确保像素的内存不会被移动。如果原

生应用程序想要访问像素数据并操作它，该方法返回了像素缓存的一个原生指针。

```
int AndroidBitmap_lockPixels(JNIEnv* env,
                             jobject jbitmap,
                             void** addrPtr);
```

该函数的参数为 JNIEnv 接口指针、Bitmap 对象的引用和 void 指针，用来返回原生像素缓存的地址。成功的话返回 0；否则返回一个负数。和 AndroidBitmap_getInfo 函数一样，AndroidBitmap_lockPixels 函数的全部错误码列表可以在 android/bitmap.h 头文件中找到。

3. 释放原生像素缓存

对 AndroidBitmap_lockPixels 的每次调用都应该对应一次 AndroidBitmap_unlockPixels 调用，用来释放原生像素缓存。当完成对原生像素缓存的读写时，原生应用程序应该释放它。一旦释放，Bitmap 对象就可以在 Java 层使用了。

```
int AndroidBitmap_unlockPixels(JNIEnv* env,jobject jbitmap);
```

AndroidBitmap_unlockPixels 函数的参数是一个 JNIEnv 接口指针和 Bitmap 对象的引用。成功的话，返回 0；否则，返回一个负数。

12.3.3　用 Bitmap 渲染来更新 AVI Player

想要修改 AVI player，可以执行以下步骤：

(1) 打开 Project Explorer，打开 AndroidManifest.xml 清单文件并且声明一个新的 activity，如程序清单 12-15 所示。

程序清单 12-15　Manifest 文件中声明的新 Bitmap Player Activity

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.apress.aviplayer"
    android:versionCode="1"
    android:versionName="1.0" >

    ...

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

        ...

        <activity
            android:name=".BitmapPlayerActivity"
            android:label="@string/title_activity_bitmap_player" >
        </activity>
```

```
    </application>

</manifest>
```

(2) 新Bitmap Player activity 的标题以及Bitmap Player 单选按钮的标识都应该添加到字符串资源中。打开 strings.xml 字符串资源文件，并添加新的字符串资源，如程序清单 12-16 所示。

程序清单 12-16　所添加的 Bitmap Player activity 字符串资源

```
<resources>

    ...
    <string name="bitmap_player_radio">Bitmap Player</string>
    <string name="title_activity_bitmap_player">Bitmap Player</string>

</resources>
```

(3) Bitmap Player activity 想要运行的话，需要一个单独的 SurfaceView 小控件。打开 Project Explorer，展开 res 目录。

(4) 右击 layout 子目录，在上下文菜单中选择 New | File。

(5) 将 File Name 设置为 activity_bitmap_player.xml。

(6) 用程序清单 12-17 的内容替换新布局文件内容。

程序清单 12-17　activity_bitmap_player.xml 布局文件的内容

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <SurfaceView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/surface_view" />

</LinearLayout>
```

(7) 打开 Project Explorer，展开 src 目录。

(8) 右击 com.apress.aviplayer 包名，在上下文菜单中选择 New | Class。

(9) 将 Name 设置为 BitmapPlayerActivity。

(10) 单击 Finish 按钮创建一个新类。

(11) 用程序清单 12-18 的内容替换新类的内容。

程序清单 12-18　BitmapPlayerActivity 源文件的内容

```
package com.apress.aviplayer;
```

```
import java.util.concurrent.atomic.AtomicBoolean;

import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.os.Bundle;
import android.view.SurfaceHolder;
import android.view.SurfaceHolder.Callback;
import android.view.SurfaceView;

/**
 *使用 bitmap 的 AVI player
 *
 * @author Onur Cinar
 */
    public class BitmapPlayerActivity extends AbstractPlayerActivity {
        /** 正在播放*/
        private final AtomicBoolean isPlaying = new AtomicBoolean();

        /** Surface holder. */
        private SurfaceHolder surfaceHolder;

        /**
         * 创建过程
         *
         * @param savedInstanceState saved state.
         */
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_bitmap_player);

            SurfaceView surfaceView = (SurfaceView)
                    findViewById(R.id.surface_view);

            surfaceHolder = surfaceView.getHolder();
            surfaceHolder.addCallback(surfaceHolderCallback);
    }

    /**
     * Surface holder 监听 surface 事件的回调
     */
    private final Callback surfaceHolderCallback = new Callback() {
        public void surfaceChanged(SurfaceHolder holder, int format,
                int width, int height) {
        }

        public void surfaceCreated(SurfaceHolder holder) {
            //surface 准备好后开始播放
            isPlaying.set(true);
```

```
            //在一个单独的线程中渲染
            new Thread(renderer).start();
        }

        public void surfaceDestroyed(SurfaceHolder holder) {
            // Surface 销毁后停止播放
            isPlaying.set(false);
        }
    };

    /**
     * 渲染线程通过一个 bitmap 将 AVI 文件中的视频帧渲染到 surface 上
     */
    private final Runnable renderer = new Runnable() {
        public void run() {
            // 创建一个新的 bitmap 来保存所有的帧
            Bitmap bitmap = Bitmap.createBitmap(
                    getWidth(avi),
                    getHeight(avi),
                    Bitmap.Config.RGB_565);

            // 使用帧速来计算延迟
            long frameDelay = (long) (1000 / getFrameRate(avi));

            // 播放的时候开始渲染
            while (isPlaying.get()) {
                // 将帧渲染至 bitmap
                render(avi, bitmap);

                //锁定 canvas
                Canvas canvas = surfaceHolder.lockCanvas();

                //将 bitmap 绘制至 canvas
                canvas.drawBitmap(bitmap, 0, 0, null);

                // canvas 准备显示
                surfaceHolder.unlockCanvasAndPost(canvas);

                //等待下一帧
                try {
                    Thread.sleep(frameDelay);
                } catch (InterruptedException e) {
                    break;
                }
            }
        }
    };

    /**
     * 从 AVI 文件描述符输出到指定 Bitmap 来渲染帧
```

```
         *
         * @param avi file descriptor.
         * @param bitmap bitmap instance.
         * @return true if there are more frames, false otherwise.
         */
        private native static boolean render(long avi, Bitmap bitmap);
    }
```

BitmapPlayerActivity 通过一个叫做 render 的原生方法来处理视频帧的渲染。

(12) 在顶部菜单栏中选择 Project | Build 来编译 Java 源代码。

(13) 打开 Project Explorer 选择 BitmapPlayerActivity。

(14) 在主菜单栏中选择 Run | External Tools | Generate C and C++ Header File 来对 BitmapPlayerActivity 类调用 javah 工具。

(15) 在项目的 jni 子目录下，javah 工具将生成一个 com_apress_aviplayer_ BitmapPlayerActivity.h 头文件。

(16) 右击 jni 目录，在上下文菜单中选择 New | Source File。

(17) 将 Source File 设置为 com_apress_aviplayer_BitmapPlayerActivity.cpp。

(18) 单击 Finish 按钮来创建一个新的 C++源文件。

(19) 打开 Eclipse，用程序清单 12-19 的内容替换新源文件的内容。

程序清单 12-19　com_apress_aviplayer_BitmapPlayerActivity.cpp 的内容

```
extern "C" {
#include <avilib.h>
}

#include <android/bitmap.h>

#include "Common.h"
#include "com_apress_aviplayer_BitmapPlayerActivity.h"

jboolean Java_com_apress_aviplayer_BitmapPlayerActivity_render(
        JNIEnv* env,
        jclass clazz,
        jlong avi,
        jobject bitmap)
{
    jboolean isFrameRead = JNI_FALSE;

    char* frameBuffer = 0;
    long frameSize = 0;
    int keyFrame = 0;

    //锁定 bitmap 并得到 raw byte
    if (0 > AndroidBitmap_lockPixels(env, bitmap, (void**) &frameBuffer))
    {
        ThrowException(env, "java/io/IOException",
                "Unable to lock pixels.");
```

```
            goto exit;
        }

        // 将 AVI 帧 byte 读到 bitmap 中
        frameSize = AVI_read_frame((avi_t*) avi, frameBuffer, &keyFrame);

        // 解锁 bitmap
        if (0 > AndroidBitmap_unlockPixels(env, bitmap))
        {
            ThrowException(env, "java/io/IOException",
                    "Unable to unlock pixels.");
            goto exit;
        }

        // 检查帧是否成功读取
        if (0 < frameSize)
        {
            isFrameRead = JNI_TRUE;
        }

    exit:
        return isFrameRead;
    }
```

(20) 需要修改构建文件 Android.mk 来编译新的源文件以及动态链接 jnigraphics 共享库来使用 JNI Graphics Bitmap API，如程序清单 12-20 所示。

程序清单 12-20　修改 Bitmap Player 的构建文件

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := AVIPlayer
LOCAL_SRC_FILES := \
    Common.cpp \
    com_apress_aviplayer_AbstractPlayerActivity.cpp \
    com_apress_aviplayer_BitmapPlayerActivity.cpp

# Use AVILib static library
LOCAL_STATIC_LIBRARIES += avilib_static

# Link with JNI graphics
LOCAL_LDLIBS += -ljnigraphics

include $(BUILD_SHARED_LIBRARY)

# Import AVILib library module
$(call import-module, transcode-1.1.5/avilib)
```

(21) bitmap player activity 现在准备好了。想要使用它的话，需要把它作为一个单选按钮添加到 activity_main.xml 布局文件中，如程序清单 12-21 所示。

程序清单 12-21　将 Bitmap Player 单选按钮添加到 Main Activity 布局中

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ...

    <RadioGroup
        android:id="@+id/player_radio_group"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >

        <RadioButton
            android:id="@+id/bitmap_player_radio"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:checked="true"
            android:text="@string/bitmap_player_radio" />

    </RadioGroup>

    ...

</LinearLayout>
```

(22) main activity 的源代码也需要修改，如程序清单 12-22 所示，当用户选择时，需要向 Bitmap Player activity 发送播放请求。

程序清单 12-22　将 Bitmap Player 单选按钮事件添加到 Main Activity 中

```
/**
 * play 按钮单击事件处理
 */
private void onPlayButtonClick() {

    ...

    // 基于 id 选择 activity
    switch (radioId) {
    case R.id.bitmap_player_radio:
        intent = new Intent(this, BitmapPlayerActivity.class);
        break;

    default:
```

```
            throw new UnsupportedOperationException("radioId=" + radioId);
        }

        ...
    }
```

12.3.4 运行使用 Bitmap 渲染的 AVI Player

现在，基于 JNI 图形 API 使用 Bitmap 渲染的 AVI Player 应用程序准备好了。按照以下步骤执行在 Android 模拟器上测试应用程序。

(1) 想要测试这个 AVI 视频播放器应用程序，需要一个 AVI 格式的视频文件。为了简化示例，应用程序只使用 AVI 格式作为容器，并希望视频负载是通过 RGB565 颜色空间的未压缩原始帧来提供的。用你最喜欢的浏览器从作者的网站 http://zdo.com/galleon.zip 下载该示例视频文件。

(2) 提取下载的 ZIP 压缩包中的 galleon.avi AVI 视频文件。

(3) 启动 Android 模拟器。

(4) 使用 ADB，将 AVI 视频文件推送到 Android 模拟器的 SD 卡中，如下：

```
adb push galleon.avi /sdcard/
```

注意：

galleon.avi AVI 视频文件需要 SD 卡上至少有 74MB 的剩余空间。如果 ADB push 文件出错，请确保 Android 设备或者 Android 模拟器有足够的空间。由于视频文件较大，将它推送到 SD 卡上要花费 30 秒或者更多的时间。

(5) 在 Android 模拟器上启动 AVI player 应用程序。

(6) 确保已经选中 Bitmap Player 单选按钮，如图 12-1 所示：

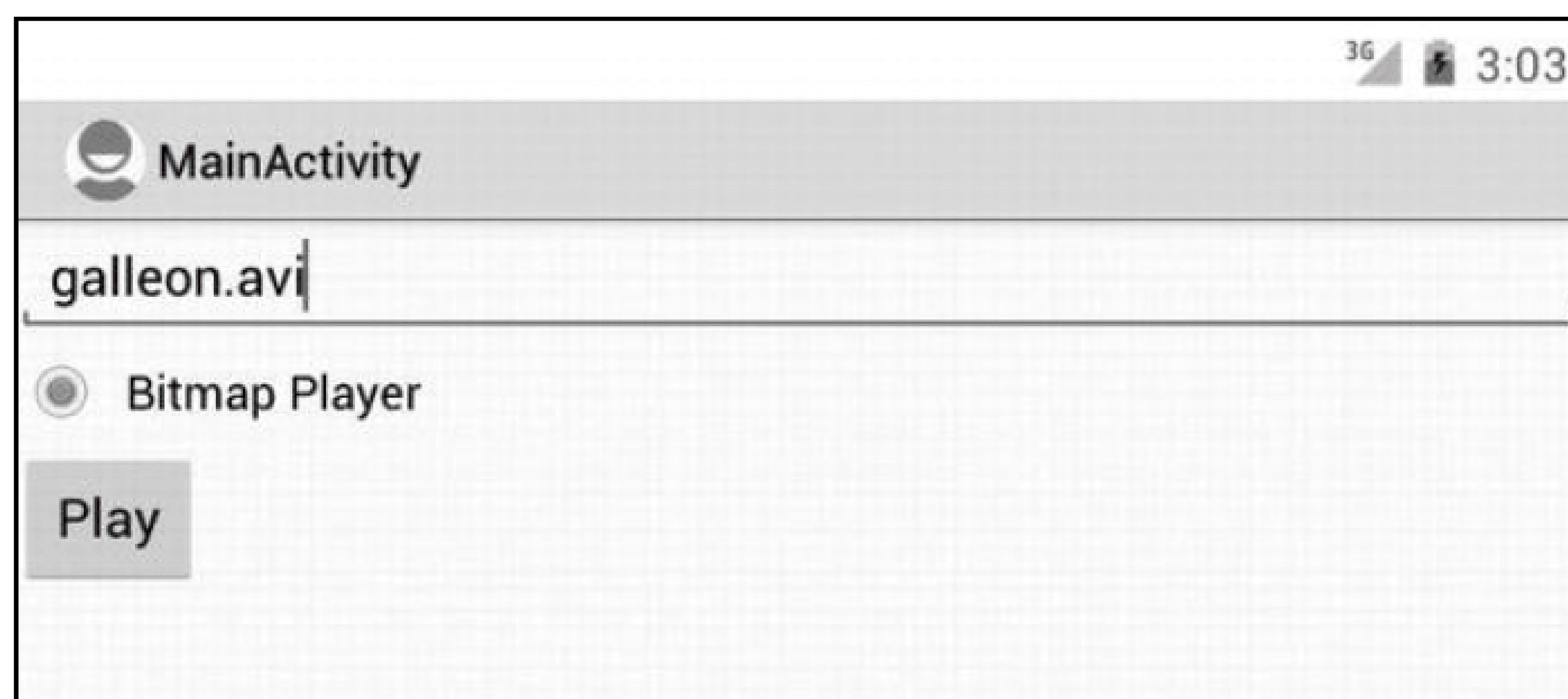


图 12-1　使用 AVI 播放器 GUI 来选择 Bitmap 播放器

(7) 单击 Play 按钮开始播放。Bitmap Player activity 将会被调用并且 AVI 视频文件将通过 JNI Graphics API 渲染，如图 12-2 所示。你将看到帆船上飘扬的白旗。

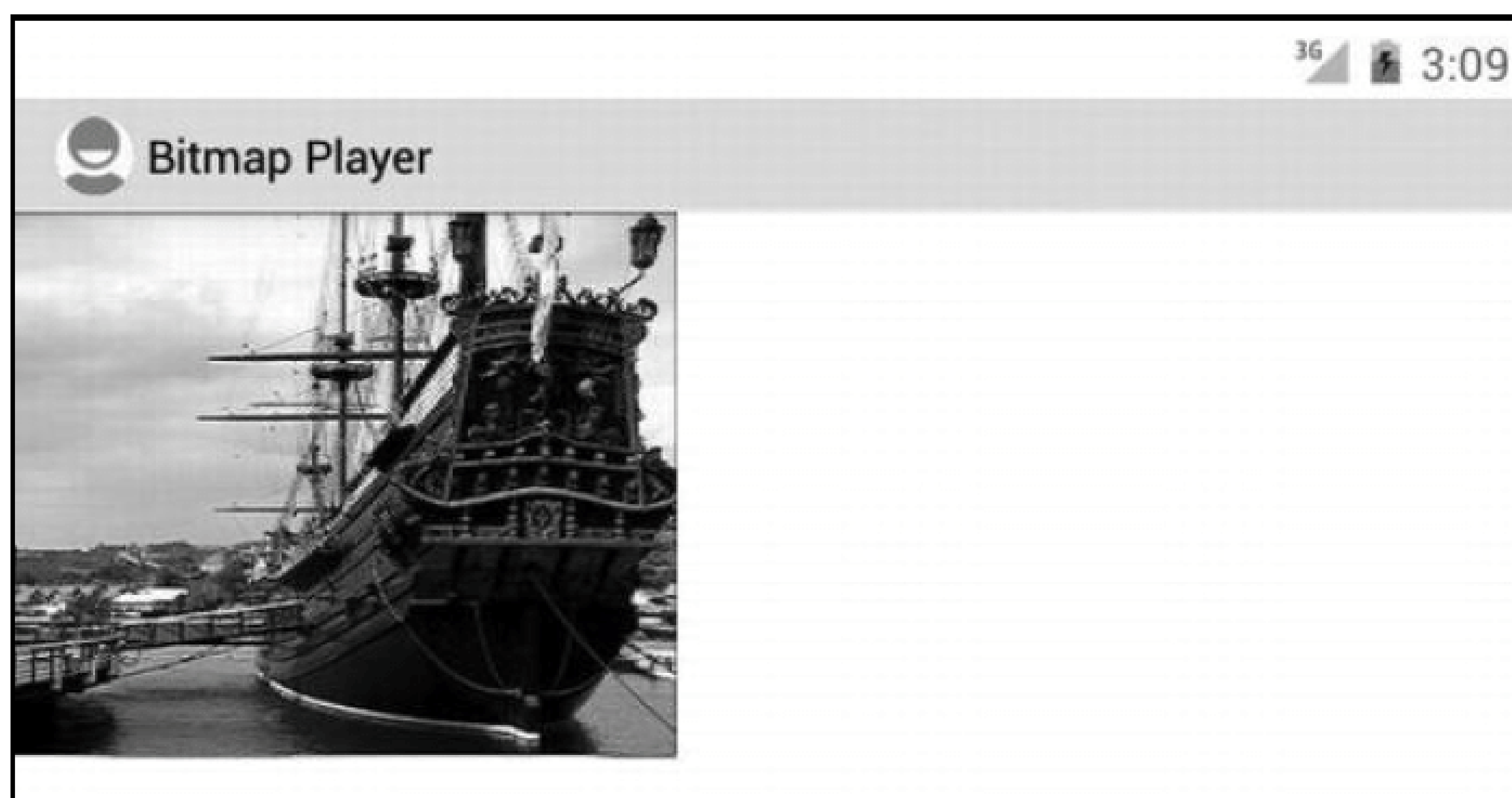


图 12-2　通过 Bitmap 渲染程序渲染 AVI 视频文件

12.4　使用 OpenGL ES 渲染

Android NDK 为原生代码提供了 OpenGL ES 的 1.x 和 2.0 图形 API，正如本章前面所示。

- Android 1.6 及以后版本支持的 OpenGL ES 1.0
- 只在带有相应的 GPU 的特殊设备上支持 OpenGL ES 1.1
- Android 2.0 及以后版本支持的 OpenGL ES 2.0

应用程序应该在 Android 清单文件中使用<uses-feature>标签来标识要使用的 OpenGL ES 的首选版本。

12.4.1　使用 OpenGL ES API

想要使用 OpenGL ES API，需要在 Java 代码中添加一个 android.opengl.GLSurfaceView 实例。这样，原生应用程序就可以调用 OpenGL ES API 函数向 GLSurfaceView 渲染图像。更多关于可用 OpenGL ES API 的信息可以在 Khronos Group 网站 www.khronos.org/opengles/ 上查询到。

本书编写时，Android 模拟器还不支持 OpenGL ES 2.0 硬件模拟。为了体验基于 OpenGL ES 的图形 API，示例应用程序将使用 OpenGL ES 1.x。

12.4.2　启用 OpenGL ES 1.x API

执行以下步骤在原生应用程序中使用 OpenGL ES 1.x。

(1) 包含 OpenGL ES 1.x 头文件。

```
#include <GLES/gl.h>
#include <GLES/glext.h>
```

(2) 修改 Android.mk 构建文件来动态链接 GLESv1_CM 库。

```
LOCAL_LDLIBS += -lGLESv1_CM
```

做了这些修改后，在原生应用程序中就可以使用 OpenGL ES 1.x API 了。

12.4.3 启用 OpenGL ES 2.0 API

执行以下步骤在原生应用程序中使用 OpenGL ES 2.0。

(1) 包含 OpenGL ES 2.0 头文件。

```
#include <GLES2/gl2.h>
#include <GLES2/gl2ext.h>
```

(2) 修改 Android.mk 构建文件来动态链接 GLESv2 库。

```
LOCAL_LDLIBS += -lGLESv2
```

做了这些修改后，在原生应用程序中就可以使用 OpenGL ES 2.0 API 了。

12.4.4 用 OpenGL ES 渲染来更新 AVI Player

执行以下步骤。

(1) 打开 Project Explorer，打开 AndroidManifest.xml 清单文件并声明新的 activity，如程序清单 12-23 所示。

程序清单 12-23 在清单文件中声明新的 OpenGL Player Activity

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.apress.aviplayer"
    android:versionCode="1"
    android:versionName="1.0" >

    ...

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

        ...

        <activity
            android:name=".OpenGLPlayerActivity"
            android:label="@string/title_activity_open_gl_player" >
        </activity>
    </application>

</manifest>
```

(2) 和 OpenGL player 单选按钮一样，新的 OpenGL player activity 的标题应该添加到字

符串资源文件中。打开 strings.xml 字符串资源文件并添加新的字符串资源，如程序清单 12-24 所示。

程序清单 12-24　添加 OpenGL Player Activity 字符串资源

```
<resources>

    ...

    <string name="title_activity_open_gl_player">OpenGL Player</string>
    <string name="open_gl_player_radio">OpenGL Player</string>
</resources>
```

(3) 想要运行 Bitmap Player activity 的话，需要一个单独的 GLSurfaceView 小控件。打开 Project Explorer，展开 res 目录。

(4) 右击 layout 子目录，在上下文菜单中选择 New | File。

(5) 将 File Name 设置为 activity_open_gl_player.xml。

(6) 用程序清单 12-25 的内容替换新的布局文件内容。

程序清单 12-25　activity_open_gl_player.xml 布局文件的内容

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <android.opengl.GLSurfaceView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/gl_surface_view" />

</LinearLayout>
```

(7) 打开 Project Explorer，展开 src 目录。

(8) 右击 com.apress.aviplayer 包名，在上下文菜单中选择 New | Class。

(9) 设置 Name 为 OpenGLPlayerActivity。

(10) 单击 Finish 按钮来创建新的类。

(11) 使用程序清单 12-26 的内容替换它的内容。

程序清单 12-26　OpenGLPlayerActivity.java 源文件的内容

```
package com.apress.aviplayer;

import java.util.concurrent.atomic.AtomicBoolean;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;
```

```
import android.opengl.GLSurfaceView;
import android.opengl.GLSurfaceView.Renderer;
import android.os.Bundle;

/**
 * 使用 OpenGL 的 AVI 播放器.
 *
 * @author Onur Cinar
 */
public class OpenGLPlayerActivity extends AbstractPlayerActivity {
    /** */正在播放
    private final AtomicBoolean isPlaying = new AtomicBoolean();

    /** 原生渲染器*/
    private long instance;

    /** GL surface view 实例*/
    private GLSurfaceView glSurfaceView;

    /**
     * On create.
     *
     * @param savedInstanceState saved state.
     */
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_open_gl_player);

        glSurfaceView = (GLSurfaceView)
                findViewById(R.id.gl_surface_view);

        // 设置渲染器
        glSurfaceView.setRenderer(renderer);

        // 请求时渲染帧
        glSurfaceView.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    }

    /**
     * On start.
     */
    protected void onStart() {
        super.onStart();

        // 初始化原生渲染器
        instance = init(avi);
    }

    /**
     * On resume.
```

```
 */
protected void onResume() {
    super.onResume();

    // 当 activity resumed 时必须通知 GL surface 视图
    glSurfaceView.onResume();
}

/**
 * On pause.
 */
protected void onPause() {
    super.onPause();

    // 当 activity paused 时必须通知 GL surface view
    glSurfaceView.onPause();
}

/**
 * On stop.
 */
protected void onStop() {
    super.onStop();

    //释放原生渲染器
    free(instance);
    instance = 0;
}

/**
 *根据帧速请求渲染
 */
private final Runnable player = new Runnable() {
    public void run() {
        //使用帧速计算延迟
        long frameDelay = (long) (1000 / getFrameRate(avi));

        // 播放时开始渲染
        while (isPlaying.get()) {
            // 请求渲染
            glSurfaceView.requestRender();

            // 等待下一帧
            try {
                Thread.sleep(frameDelay);
            } catch (InterruptedException e) {
                break;
            }
        }
    }
```

```
};

/**
 * OpenGL renderer.
 */
private final Renderer renderer = new Renderer() {
    public void onDrawFrame(GL10 gl) {
        // 渲染下一帧
        if (!render(instance, avi))
        {
            isPlaying.set(false);
        }
    }

    public void onSurfaceChanged(GL10 gl, int width, int height) {

    }

    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        //初始化 OpenGL surface
        initSurface(instance, avi);

        // surface 准备好后开始播放
        isPlaying.set(true);

        // 启动播放器
        new Thread(player).start();
    }
};

/**
 *初始化原生渲染器
 *
 * @param avi file descriptor.
 * @return native instance.
 */
private native static long init(long avi);

/**
 *初始化 OpenGL surface
 *
 * @param instance native instance.
 */
private native static void initSurface(long instance, long avi);

/**
 *用给定文件进行帧渲染
 *
 * @param instance native instance.
 * @param avi file descriptor.
```

```
     * @return true if there are more frames, false otherwise.
     */
    private native static boolean render(long instance, long avi);

    /**
     * 释放原生渲染器
     *
     * @param instance native instance.
     */
    private native static void free(long instance);
}
```

(12) 从顶部菜单栏中选择 Project | Build Project 编译 Java 源代码。

(13) 打开 Project Explorer，选择 OpenGLPlayerActivity。

(14) 在主菜单中选择 Run | External Tools | Generate C and C++ Header File 对 OpenGLPlayerActivity 类调用 javah 工具。

(15) 在项目的 jni 子目录下，javah 工具会生成 com_apress_aviplayer_OpenGLPlayerActivity.h 头文件。

(16) 右击 jni 目录，在上下文菜单中选择 New | Source File。

(17) 将 Source File 设置为 com_apress_aviplayer_OpenGLPlayerActivity.cpp。

(18) 单击 Finish 按钮创建一个新的 C++源文件。

(19) 打开 Eclipse，用程序清单 12-27 的内容替换新的源文件内容。

程序清单 12-27　com_apress_aviplayer_OpenGLPlayerActivity.cpp 的内容

```
extern "C" {
#include <avilib.h>
}

#include <GLES/gl.h>
#include <GLES/glext.h>
#include <malloc.h>

#include "Common.h"
#include "com_apress_aviplayer_OpenGLPlayerActivity.h"

struct Instance
{
    char* buffer;
    GLuint texture;

    Instance():
        buffer(0),
        texture(0)
    {

    }
};
```

```
jlong Java_com_apress_aviplayer_OpenGLPlayerActivity_init(
        JNIEnv* env,
        jclass clazz,
        jlong avi)
{
    Instance* instance = 0;

    long frameSize = AVI_frame_size((avi_t*) avi, 0);
    if (0 >= frameSize)
    {
        ThrowException(env, "java/io/RuntimeException",
                "Unable to get the frame size.");
        goto exit;
    }

    instance = new Instance();
    if (0 == instance)
    {
        ThrowException(env, "java/io/RuntimeException",
                "Unable to allocate instance.");
        goto exit;
    }

    instance->buffer = (char*) malloc(frameSize);
    if (0 == instance->buffer)
    {
        ThrowException(env, "java/io/RuntimeException",
                "Unable to allocate buffer.");
        delete instance;
        instance = 0;
    }

    exit:
        return (jlong) instance;
    }

    void Java_com_apress_aviplayer_OpenGLPlayerActivity_initSurface(
            JNIEnv* env,
            jclass clazz,
            jlong inst,
            jlong avi)
    {
        Instance* instance = (Instance*) inst;

        // 启用纹理
        glEnable(GL_TEXTURE_2D);

        // 生成一个纹理对象
        glGenTextures(1, &instance->texture);
```

```
    //绑定到生成的纹理上
    glBindTexture(GL_TEXTURE_2D, instance->texture);

    int frameWidth = AVI_video_width((avi_t*) avi);
    int frameHeight = AVI_video_height((avi_t*) avi);

    // 剪切纹理矩形
    GLint rect[] = {0, frameHeight, frameWidth, -frameHeight};
    glTexParameteriv(GL_TEXTURE_2D, GL_TEXTURE_CROP_RECT_OES, rect);

    //填充颜色
    glColor4f(1.0, 1.0, 1.0, 1.0);

    //生成一个空的纹理
    glTexImage2D(GL_TEXTURE_2D,
        0,
        GL_RGB,
        frameWidth,
        frameHeight,
        0,
        GL_RGB,
        GL_UNSIGNED_SHORT_5_6_5,
        0);
}

jboolean Java_com_apress_aviplayer_OpenGLPlayerActivity_render(
        JNIEnv* env,
        jclass clazz,
        jlong inst,
        jlong avi)
{
    Instance* instance = (Instance*) inst;
    jboolean isFrameRead = JNI_FALSE;
    int keyFrame = 0;

    //将 AVI 帧字节读至 bitmap
    long frameSize = AVI_read_frame((avi_t*) avi,
            instance->buffer,
            &keyFrame);

    // 检查帧是否读了
    if (0 >= frameSize)
    {
        goto exit;
    }

    // 读帧
    isFrameRead = JNI_TRUE;
```

```
        //使用新帧更新纹理
        glTexSubImage2D(GL_TEXTURE_2D,
                0,
                0,
                0,
                AVI_video_width((avi_t*) avi),
                AVI_video_height((avi_t*) avi),
                GL_RGB,
                GL_UNSIGNED_SHORT_5_6_5,
                instance->buffer);

        // 绘制纹理
        glDrawTexiOES(0, 0, 0,
                AVI_video_width((avi_t*) avi),
                AVI_video_height((avi_t*) avi));
    exit:
        return isFrameRead;
    }

    void Java_com_apress_aviplayer_OpenGLPlayerActivity_free(
            JNIEnv* env,
            jclass clazz,
            jlong inst)
    {
        Instance* instance = (Instance*) inst;

        if (0 != instance)
        {
            free(instance->buffer);
            delete instance;
        }
    }
```

(20) 需要修改构建文件 Android.mk 来编译新的源文件以及动态链接 GLESv1_CM 共享库，从而可以使用原生空间的 OpenGL ES API，如程序清单 12-28 所示。

程序清单 12-28　OpenGL Player 构建文件的修改

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := AVIPlayer
LOCAL_SRC_FILES := \
    Common.cpp \
    com_apress_aviplayer_AbstractPlayerActivity.cpp \
    com_apress_aviplayer_BitmapPlayerActivity.cpp \
    com_apress_aviplayer_OpenGLPlayerActivity.cpp

# 使用 AVILib 静态库
```

```
LOCAL_STATIC_LIBRARIES += avilib_static

...

# 启用 GL ext 原型
LOCAL_CFLAGS += -DGL_GLEXT_PROTOTYPES

# 连接 OpenGL ES
LOCAL_LDLIBS += -lGLESv1_CM

include $(BUILD_SHARED_LIBRARY)

...
```

(21) 现在 Bitmap Player activity 准备好了。想要使用它的话，需要把它作为一个单选按钮添加到 activity_main.xml 布局文件中，如程序清单 12-29 所示。

程序清单 12-29　将 OpenGL Player 单选按钮添加到 Main 布局中

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ...

    <RadioGroup
        android:id="@+id/player_radio_group"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >

        <RadioButton
            android:id="@+id/bitmap_player_radio"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:checked="true"
            android:text="@string/bitmap_player_radio" />

        <RadioButton
            android:id="@+id/open_gl_player_radio"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/open_gl_player_radio" />

    </RadioGroup>

    ...

</LinearLayout>
```

(22) main activity 的源代码也需要修改，如程序清单 12-30 所示，当用户选择时，需要向 Bitmap Player activity 发送播放请求。

程序清单 12-30　将 OpenGL Player 单选按钮事件添加到 Main Activity 中

```
/**
 * 在播放按钮上单击事件处理.
 */
private void onPlayButtonClick() {

    ...

    // 基于 id 选择 activity
    switch (radioId) {
    case R.id.bitmap_player_radio:
        intent = new Intent(this, BitmapPlayerActivity.class);
        break;

    case R.id.open_gl_player_radio:
        intent = new Intent(this, OpenGLPlayerActivity.class);
        break;

    default:
        throw new UnsupportedOperationException("radioId=" + radioId);
    }

    ...
}
```

(23) 现在，AVI player 应用程序的 OpenGL ES 渲染器准备好了。采用和本章 JNI Graphics API 一节一样的步骤在 Android 模拟器上运行示例应用程序。

12.5　使用原生 Window API 进行渲染

从 Android API level 9 开始，Android NDK 提供了一个 API 启用原生代码来直接访问和处理原生 window 的像素缓存。这个 API 称为原生 window API。本节将学习如何使用这个 API 直接在原生代码中进行渲染而无须引用任何基于 Java 的 API。

12.5.1　启用原生 Window API

执行以下步骤在应用程序中使用原生 Window API。

(1) 包含原生 window 头文件。

```
#include <android/native_window.h>
#include <android/native_window_jni.h>
```

(2) 修改 Android.mk 构建文件来动态链接 android 库。

```
LOCAL_LDLIBS += -landroid
```

做了以上修改后，原生应用程序中就可以使用原生 window API 了。

12.5.2　使用原生 Window API

原生 Window API 提供了 4 个原生函数用于访问和操作 Bitmap 对象。

1. 从 Surface 对象中检索原生 window

ANativeWindow_fromSurface 函数可以从给定的 Surface 对象中检索原生 window。

```
ANativeWindow* ANativeWindow_fromSurface(JNIEnv* env,
                                         jobject surface);
```

它的参数为一个 JNIEnv 接口指针和一个 Surface 对象引用，返回值是一个指向原生 window 实例的指针。ANativeWindow_fromSurface 同时获得了返回的原生 window 实例的引用，为了防止内容泄露，需要使用 ANativeWindow_release 函数来释放掉。

2. 获取原生 Window 实例中的引用

为了防止原生 window 实例被删除，原生代码可以使用 ANativeWindow_acquire 函数来获取它的引用。

```
void ANativeWindow_acquire(ANativeWindow* window);
```

任何 ANativeWindow_acquire 函数调用都要对应一个 ANativeWindow_release 函数的调用。

3. 释放原生 Window 引用

如上所述，为了防止内存泄露，每个原生 window 的引用都应该使用 ANativeWindow_release 函数来进行释放。

```
void ANativeWindow_release(ANativeWindow* window);
```

ANativeWindow_release 的参数为一个原生 window 实例指针。

4. 检索原生 Window 信息

原生 Window API 为原生代码提供了获取很多有关原生 window 信息的函数，例如大小和像素格式。

- ANativeWindow_getWidth 函数用于检索原生 window 的宽度。
- ANativeWindow_getHeight 函数用于检索原生 window 的高度。
- ANativeWindow_getFormat 函数用于检索原生 window 的像素格式。

5. 设置原生 window 缓冲区的几何形状

原生 window 的大小和像素格式应该与即将渲染的图像数据匹配。如果图像数据的大

小或者像素格式不一致，可以用 ANativeWindow_ setBuffersGeometry 函数来重新配置原生 window 缓冲区。这样缓冲区就会自动缩放来匹配原生 window。

```
int32_t ANativeWindow_setBuffersGeometry(ANativeWindow* window,
                                          int32_t width,
                                          int32_t height,
                                          int32_t format);
```

该函数的参数为前面获得的原生 window 实例的指针、原生 window 缓冲区的新宽度、高度以及新的像素格式。成功的话，会返回 0。对于所有参数，如果提供的值为 0，那么参数值将被恢复到原生 window 缓冲区的基础值。

6. 访问原生 window 缓冲区

ANativeWindow_lock 函数用来锁定原生 window 缓冲区并获得一个原始像素缓冲区的指针。然后，原生代码可以使用这个指针来访问和操作像素缓冲区。

```
int32_t ANativeWindow_lock(ANativeWindow* window,
                           ANativeWindow_Buffer* outBuffer,
                           ARect* inOutDirtyBounds);
```

该函数的参数是前面获得的原生 window 实例的指针、指向 ANativeWindow_Buffer 结构体的指针以及指向可选 ARect 结构体的指针。如程序清单 12-31 所示，ANativeWindow_Buffer 结构体除了拥有原生 window 的相关信息外，还可以通过位字段来访问原生像素缓冲区。

程序清单 12-31　ANativeWindow_Buffer 结构体声明

```
typedef struct ANativeWindow_Buffer {
    // 水平显示的像素数.
    int32_t width;

    // 垂直显示的像素数
    int32_t height;

    // 缓冲区在内存中一行的像素数，可能 >= width.
    int32_t stride;

    // 缓冲区的格式. 如：WINDOW_FORMAT_*
    int32_t format;

    // 实际位数.
    void* bits;

    //不要碰
    uint32_t reserved[6];
} ANativeWindow_Buffer;
```

ANativeWindow_lock 函数执行成功的话会返回 0。

7. 释放原生 window 缓冲区

一旦原生代码完成，应该使用 ANativeWindow_unlockAndPost 函数解锁并输出原生 window 缓冲区。

```
int32_t ANativeWindow_unlockAndPost(ANativeWindow* window);
```

该函数的参数是一个已经被锁定的原生 window 实例指针。成功的话，返回 0。现在，可以更新 AVI Player 测试应用程序使用原生 window 渲染器来测试这些函数。

12.5.3 用原生 window 渲染器来更新 AVI Player

执行以下步骤。

(1) 打开 Project Explorer，打开 AndroidManifest.xml 清单文件并声明新的 Activity，如程序清单 12-32 所示。

程序清单 12-32　在清单文件中声明新的原生 window Player

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.apress.aviplayer"
    android:versionCode="1"
    android:versionName="1.0" >

    ...

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

        ...

        <activity
            android:name=".OpenGLPlayerActivity"
            android:label="@string/title_activity_open_gl_player" >
        </activity>
        <activity
            android:name=".NativeWindowPlayerActivity"
            android:label="@string/title_activity_native_window_player" >
        </activity>
    </application>

</manifest>
```

(2) 和 Bitmap Player 单选按钮一样，新的 Bitmap Player activity 的标题应该添加到字符串资源文件中。打开 strings.xml 字符串资源文件并添加新的字符串资源，如程序清单 12-33 所示。

程序清单 12-33　添加原生 window Player Activity 字符串资源

```
<resources>

    ...

    <string name="title_activity_native_window_player"
            >原生 window Player</string>
    <string name="native_window_player_radio"
            >原生 window Player</string>

</resources>
```

(3) Bitmap Player activity 想要运行的话，需要一个单独的 SurfaceView 小控件。打开 Project Explorer，展开 res 目录。

(4) 右击 layout 子目录，在上下文菜单中选择 New | File。

(5) 将 File Name 设置为 activity_native_window_player.xml。

(6) 用程序清单 12-34 的内容替换新的布局文件的内容。

程序清单 12-34　activity_native_window_player.xml 布局文件的内容

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <SurfaceView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/surface_view" />

</LinearLayout>
```

(7) 打开 Project Explorer，展开 src 目录。

(8) 右击 com.apress.aviplayer 包，在上下文菜单中选择 New | Class。

(9) 将 Name 设置为 NativeWindowPlayerActivity。

(10) 单击 Finish 按钮来创建新的类。

(11) 用程序清单 12-35 的内容替换它的内容。

程序清单 12-35　OpenGLPlayerActivity.java 源文件的内容

```
package com.apress.aviplayer;

import java.util.concurrent.atomic.AtomicBoolean;

import android.os.Bundle;
import android.view.Surface;
import android.view.SurfaceHolder;
import android.view.SurfaceHolder.Callback;
```

```
import android.view.SurfaceView;

/**
* 使用原生窗口的 AVI 播放器
*
* @author Onur Cinar
*/
public class NativeWindowPlayerActivity extends AbstractPlayerActivity {
    /** 正在播放*/
    private final AtomicBoolean isPlaying = new AtomicBoolean();

    /** Surface 存储器. */
    private SurfaceHolder surfaceHolder;

    /**
     * On create.
     *
     * @param savedInstanceState saved state.
     */
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_bitmap_player);

        SurfaceView surfaceView = (SurfaceView)
                findViewById(R.id.surface_view);

        surfaceHolder = surfaceView.getHolder();
        surfaceHolder.addCallback(surfaceHolderCallback);
    }

    /**
     * Surface holder 监听 surface 事件的回调
     */
    private final Callback surfaceHolderCallback = new Callback() {
        public void surfaceChanged(SurfaceHolder holder, int format,
                int width,
                int height) {
        }

        public void surfaceCreated(SurfaceHolder holder) {
            // surface 准备好后开始播放
            isPlaying.set(true);

            //在一个单独的线程中启动渲染器
            new Thread(renderer).start();
        }

        public void surfaceDestroyed(SurfaceHolder holder) {
            //surface 销毁时停止播放
            isPlaying.set(false);
```

```
        }
    };

    /**
     * 渲染线程通过一个 bitmap 将 AVI 文件中的视频帧渲染到 surface 上
     */
    private final Runnable renderer = new Runnable() {
        public void run() {
            //获得 surface 实例
            Surface surface = surfaceHolder.getSurface();

            //初始化原生 window
            init(avi, surface);

            //使用帧速计算延迟
            long frameDelay = (long) (1000 / getFrameRate(avi));

            //播放时开始渲染
            while (isPlaying.get()) {
                // 将帧渲染至 surface
                render(avi, surface);

                // 等待下一帧
                try {
                    Thread.sleep(frameDelay);
                } catch (InterruptedException e) {
                    break;
                }
            }
        }
    };

    /**
     *初始化原生 window
     *
     * @param avi file descriptor.
     * @param surface surface instance.
     */
    private native static void init(long avi, Surface surface);

    /**
     * 将给定 AVI 文件的帧渲染到给定的 surface 上
     *
     * @param avi file descriptor.
     * @param surface surface instance.
     * @return true if there are more frames, false otherwise.
     */
    private native static boolean render(long avi, Surface surface);
}
```

(12) 从主菜单中选择 Project | Build Project 来编译 Java 源代码。

(13) 打开 Project Explorer，选择 NativeWindowPlayerActivity。

(14) 在主菜单中选择 Run | External Tools | Generate C and C++ Header File 对 NativeWindowPlayerActivity 类调用 javah 工具。

(15) 在项目的 jni 子目录下，javah 工具会生成 com_apress_aviplayer_NativeWindowPlayerActivity.h 头文件。

(16) 右击 jni 目录，在上下文菜单中选择 New | Source File。

(17) 将 Source File 设置为 com_apress_aviplayer_NativeWindowPlayerActivity.cpp。

(18) 单击 Finish 按钮创建一个新的 C++源文件。

(19) 打开 Eclipse，用程序清单 12-36 的内容替换新的源文件的内容。

程序清单 12-36　com_apress_aviplayer_NativeWindowPlayerActivity.cpp 的内容

```
extern "C" {
#include <avilib.h>
}

#include <android/native_window_jni.h>
#include <android/native_window.h>

#include "Common.h"
#include "com_apress_aviplayer_NativeWindowPlayerActivity.h"

void Java_com_apress_aviplayer_NativeWindowPlayerActivity_init(
        JNIEnv* env,
        jclass clazz,
        jlong avi,
        jobject surface)
{
    //从 surface 中获得原生 window
    ANativeWindow* nativeWindow = ANativeWindow_fromSurface(
            env, surface);
    if (0 == nativeWindow)
    {
        ThrowException(env, "java/io/RuntimeException",
                "Unable to get native window from surface.");
        goto exit;
    }

    //设置 Buffer 大小为 AVI 视频帧的分辨率
    //如果和 window 的物理大小不一致
    // Buffer 会被缩放来匹配这个大小
    if (0 > ANativeWindow_setBuffersGeometry(nativeWindow,
            AVI_video_width((avi_t*) avi),
            AVI_video_height((avi_t*) avi),
            WINDOW_FORMAT_RGB_565))
    {
        ThrowException(env, "java/io/RuntimeException",
```

```
                "Unable to set buffers geometry.");
    }

    //释放原生 window
    ANativeWindow_release(nativeWindow);
    nativeWindow = 0;

exit:
    return;
}

jboolean Java_com_apress_aviplayer_NativeWindowPlayerActivity_render(
        JNIEnv* env,
        jclass clazz,
        jlong avi,
        jobject surface)
{
    jboolean isFrameRead = JNI_FALSE;

    long frameSize = 0;
    int keyFrame = 0;

    // 从 surface 中获取原生 window
    ANativeWindow* nativeWindow = ANativeWindow_fromSurface(
            env, surface);
    if (0 == nativeWindow)
    {
        ThrowException(env, "java/io/RuntimeException",
                "Unable to get native window from surface.");
        goto exit;
    }

    //锁定原生 window 并访问原始 Buffer
    ANativeWindow_Buffer windowBuffer;
    if (0 > ANativeWindow_lock(nativeWindow, &windowBuffer, 0))
    {
        ThrowException(env, "java/io/RuntimeException",
                "Unable to lock native window.");
        goto release;
    }

    //将 AVI 帧的比特流读至原始缓冲区
    frameSize = AVI_read_frame((avi_t*) avi,
            (char*) windowBuffer.bits,
            &keyFrame);

    // 检查帧是否被成功读取
    if (0 < frameSize)
    {
        isFrameRead = JNI_TRUE;
```

```
    }

    // 解锁并且输出缓冲区来显示
    if (0 > ANativeWindow_unlockAndPost(nativeWindow))
    {
        ThrowException(env, "java/io/RuntimeException",
                "Unable to unlock and post to native window.");
        goto release;
    }

release:
    //释放原生 window
    ANativeWindow_release(nativeWindow);
    nativeWindow = 0;

exit:
    return isFrameRead;
}
```

(20) 需要修改构建文件 Android.mk 编译新的源文件以及动态链接 android 共享库，从而使用原生 window API，如程序清单 12-37 所示。

程序清单 12-37　原生 window Player 构建文件的修改

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := AVIPlayer
LOCAL_SRC_FILES := \
    Common.cpp \
    com_apress_aviplayer_AbstractPlayerActivity.cpp \
    com_apress_aviplayer_BitmapPlayerActivity.cpp \
    com_apress_aviplayer_OpenGLPlayerActivity.cpp \
    com_apress_aviplayer_NativeWindowPlayerActivity.cpp

...

# 连接 Android 库
LOCAL_LDLIBS += -landroid

include $(BUILD_SHARED_LIBRARY)

...
```

(21) Bitmap Player activity 现在就准备好了。想要使用它的话，需要把它作为一个单选按钮添加到 activity_main.xml 布局文件中，如程序清单 12-38 所示。

程序清单 12-38　将原生 window Player 单选按钮添加到 Main 布局中

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ...

    <RadioGroup
        android:id="@+id/player_radio_group"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >

        ...

        <RadioButton
            android:id="@+id/open_gl_player_radio"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/open_gl_player_radio" />

        <RadioButton
            android:id="@+id/native_window_player_radio"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/native_window_player_radio" />

    </RadioGroup>

    ...

</LinearLayout>
```

(22) main activity 的源代码也需要修改，如程序清单 12-39 所示，当用户选择时，需要向 Bitmap Player activity 发送播放请求。

程序清单 12-39　原生 window Player 单选按钮事件添加到 Main Activity 中

```
/**
 * 在播放按钮上单击事件处理器
 */
private void onPlayButtonClick() {

    ...

    // 基于 id 选择 activity
    switch (radioId) {
    case R.id.bitmap_player_radio:
```

```
                intent = new Intent(this, BitmapPlayerActivity.class);
                break;

            case R.id.open_gl_player_radio:
                intent = new Intent(this, OpenGLPlayerActivity.class);
                break;

            case R.id.native_window_player_radio:
                intent = new Intent(this, NativeWindowPlayerActivity.class);
                break;

            default:
                throw new UnsupportedOperationException("radioId=" + radioId);
            }

            ...
        }
```

(23) 现在，AVI player 应用程序的原生 window 渲染器准备好了。采用和本章“JNI Graphics API”一节一样的步骤在 Android 模拟器上运行示例应用程序。

12.5.4　EGL 图形库

从 API Level 9 开始，Android NDK 也支持 EGL 图形库，可以用原生应用程序来管理 OpenGL ES surface。更多关于 EGL 的信息可以在 Khronos Group 网站 www.khronos.org/egl 上查询到。

想要启用 EGL 图形库，执行以下步骤：

(1) 包含 EGL 头文件。

```
#include <EGL/egl.h>
#include <EGL/eglext.h>
```

(2) 修改 Android.mk 构建文件来动态链接 EGL 库。

```
LOCAL_LDLIBS += -lEGL
```

通过这些修改，现在可以在你的原生应用程序中使用 EGL 图形库了。你可以使用 EGL 图形库 API 函数来列出所支持的 EGL 配置、分配和释放 OpenGL ES surface 以及用来显示的交换/单击 surface。

12.6　小结

本章探讨了原生应用程序可用的不同原生图形 API。本章通篇构建了一个 AVI 视频播放器应用程序来让我们更好地理解这些原生图形 API。

第 13 章

原生音频 API

在第 12 章中，我们已经体会到 Android 平台提供的原生图形 API 的多姿多彩。从 Android 操作系统 2.3 版本 API level 9 开始，Android 平台也提供一系列原生音频 API，使得原生代码可以方便地播放及记录音频而无须调用 Java 层的任何函数。Android 原生音频支持基于由 Khronos Group 提供的 OpenSL ES 1.0.1 标准。OpenSL ES 是 Open Sound Library for Embedded Systems 的缩写。本章我们将简要介绍与 Android 平台有关的 OpenSL ES 原生音频 API。

13.1 使用 OpenSL ES API

鉴于 OpenSL ES 规范庞大，本章将只涉及有关 Android 平台的部分。关于 OpenSL ES 更多的信息可以在$ANDROID_NDK_HOME/docs/opensles/OpenSL_ES_Specification_1.0.1.pdf 中找到。

(1) OpenSL ES API 通过一系列头文件展示。需要被包含的主要头文件是 SLES/OpenSLES.h。

```
#include <SLES/OpenSLES.h>
```

(2) 为了使用 Android 的扩展功能，源文件中也同样应该包含头文件 SLES/OpenSLES_Android.h。

```
#include <SLES/OpenSLES_Android.h>
```

(3) OpenSL ES 原生音频 API 也需要有与原生模块动态链接的库文件。这可以通过在 Android.mk 构建脚本中添加如下代码实现：

```
LOCAL_IDLIBS += -lOpenSLES
```

Android 平台对于使用 OpenSL ES 的应用程序是二进制兼容的。只需要简单链接共享

库文件，同样的应用程序即可在提供该特性的平台上无缝运行。

13.1.1 与 OpenSL ES 标准的兼容性

尽管基于 OpenSL ES 1.0.1 规范，但 Android 原生音频 API 并没有遵循任何 OpenSL ES 文件的实现。在实现中与 Android 相关的部分通过调用 Android Extensions API 展示。更多关于 Android Extensions 的信息请参阅 Android NKD 文档，其路径如下：$ANDROID_NDK_HOME/docs/opensles/index.html。

13.1.2 音频许可

在音频许可方面，使用原生音频 API 与使用基于 Java 的音频 API 没有任何区别。应用程序需要在清单文件中使用 usespermission 标签申请获取适当的许可。

- 需要包含 android.permission.RECORD_AUDIO 以便创建一个音频录音机。
- 需要包含 android.permission.MODIFY_AUDIO_SETTINGS 以便改变音频设置及使用音效。

13.2 创建 WAVE 音频播放器

WAVE 音频播放器应用程序将作为测试平台来演示 Android 平台上基于 OpenSL ES 的原生音频播放，示例应用程序将提供以下内容：

- 支持原生代码的 Android 应用程序项目。
- 在原生代码中解析 WAVE 音频文件的静态链接 WAVE 库文件。
- 支持基于 OpenSL ES 的 WAVE 音频文件播放。
- 从 SD 卡中指定要播放的 WAVE 文件的简单 GUI。

播放 WAVE 音频文件需要解析 WAVE 文件。虽然 WAVE 格式并不十分复杂，但为方便起见，我们会用一个第三方 WAVE 库文件来处理 WAVE 文件。

注意：
关于此示例应用程序的全部源代码可以从出版商的网站上下载：www.apress.com。

13.2.1 将 WAVELib 作为 NDK 导入模块

第 10 章所用到的 AVILib 库文件也是通过 WAVELib 提供对 WAVE 音频文件的支持。通过以下步骤将 WAVELib 添加为 NDK 导入模块。

(1) 如果使用 Mac OS 或者 Linux 系统则打开一个终端窗口，如果用 Windows 系统请打开 Cygwin。

(2) 输入以下命令，将当前目录改为 AVILib(在第 10 章中已经安装)的 Android NDK 导入模块目录。

```
cd $ANDROID_NDK_HOME/sources/transcode-1.1.5/avilib
```

(3) 在 Eclipse 中打开 Android.mk 构建脚本。参照程序清单 13-1，为静态和共享 WAVELib 库文件添加导入模块描述。

程序清单 13-1　包含 WAVELib 导入模块的 Android.mk 构建文件

```
LOCAL_PATH := $(call my-dir)

...

#
# 转码 WAVLib
#

# 源文件
MY_WAVLIB_SRC_FILES := wavlib.c platform_posix.c

#包含导出目录
MY_WAVLIB_C_INCLUDES := $(LOCAL_PATH)

#
# WAVLib 静态
#
include $(CLEAR_VARS)

# 模块名称
LOCAL_MODULE := wavlib_static

# 源文件
LOCAL_SRC_FILES := $(MY_WAVLIB_SRC_FILES)

#包含导出目录
LOCAL_EXPORT_C_INCLUDES := $(MY_WAVLIB_C_INCLUDES)

# 构建静态库
include $(BUILD_STATIC_LIBRARY)

#
# WAVLib 共享
#
include $(CLEAR_VARS)

# 模块名称
LOCAL_MODULE := wavlib_shared

# 源文件
LOCAL_SRC_FILES := $(MY_WAVLIB_SRC_FILES)

#包含导出目录
LOCAL_EXPORT_C_INCLUDES := $(MY_WAVLIB_C_INCLUDES)
```

```
# 构建共享库
include $(BUILD_SHARED_LIBRARY)
```

通过以上修改，原生模块可以将 WAVELib 作为静态和共享库使用。WAVELib 现已可用于 WAVE 播放器应用程序。

13.2.2 创建 WAVE 播放器 Android 应用程序

为创建 WAVE 播放器应用程序，打开 New Android Application Project 对话框，执行以下步骤。

(1) 将 Application Name 设置为 WAV Player。

(2) 将 Project Name 设置为 WAV Player。

(3) 将 Package Name 设置为 com.apress.wavplayer。

(4) 单击 Next 按钮接受当前及后续向导页面中的默认值。

(5) 当 Android 应用程序项目创建之后，使用 Project Explorer，通过 Android Tools 弹出式菜单打开 Add Android Native Support 向导。

(6) 将 Library Name 设置为 WAVPlayer。

(7) 单击 Finish 按钮为新项目添加原生支持。

13.2.3 创建 WAVE 播放器主 Activity

主 activity 将提供一个简单的 GUI 指定从 SD 卡中选择要进行播放的 WAVE 音频文件。按如下步骤完成该主 Activity：

(1) 使用 Project Explorer 展开 res 目录资源。打开 values 子目录下 string.xml 文件，填充字符串资源，并参照程序清单 13-2 替换文件内容。

程序清单 13-2 res/values/string.xml 字符串资源文件内容

```
<resources>
    <string name="app_name">WAV Player</string>
    <string name="menu_settings">Settings</string>
    <string name="title_activity_main">MainActivity</string>
    <string name="file_name_hint">WAV file</string>
    <string name="play_button">Play</string>
    <string name="error_alert_title">Error Occurred</string>
    <string name="file_name">8k16bitpcm.wav</string>
</resources>
```

(2) 该主 activity 提供一个含有文本域的简单 GUI 来指定 WAVE 音频文件名，并通过 Play 按钮在原生代码中使用 OpenSL ES 的启动播放功能。使用 Project Explorer，展开 res 资源目录下 layout 子目录。打开 activity_main.xml 布局文件并参照程序清单 13-3 替换其内容。

程序清单 13-3　res/layout/activity_main.xml 布局文件内容

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <EditText
        android:id="@+id/fileNameEdit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="@string/file_name_hint"
        android:text="@string/file_name" >

        <requestFocus />
    </EditText>

    <Button
        android:id="@+id/playButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/play_button" />

</LinearLayout>
```

(3) 现在我们将实现该主 activity。该主 activity 启动异步播放任务，并通过调用 play 原生函数开启指定 WAVE 音频文件的播放，该函数将在本章后续部分用 OpenSL ES 实现。用 Project Explorer，打开 MainActivity.java 源文件，并参照程序清单 13-4 替换其内容。

程序清单 13-4　MainActivity.java 源文件内容

```
package com.apress.wavplayer;

import java.io.File;
import java.io.IOException;

import android.app.Activity;
import android.app.AlertDialog;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;

/**
 *
```

```
 WAVE 播放器主 activity
 *
 * @author Onur Cinar
 */
public class MainActivity extends Activity implements OnClickListener {
    /** 文件名编辑文本. */
    private EditText fileNameEdit;

    /**
     * On create.
     *
     * @param savedInstanceState
     *保存当前状态
     */
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        fileNameEdit = (EditText) findViewById(R.id.fileNameEdit);
        Button playButton = (Button) findViewById(R.id.playButton);
        playButton.setOnClickListener(this);
    }

    /**
     *单击
     * @param view
     * 视图实例
     */
    public void onClick(View view) {
        switch (view.getId()) {
        case R.id.playButton:
            onPlayButtonClick();
        }
    }

    /**
     *单击播放按钮
     */
    private void onPlayButtonClick() {
        //位于外部存储器
        File file = new File(Environment.getExternalStorageDirectory(),
                fileNameEdit.getText().toString());

        //开始播放
        PlayTask playTask = new PlayTask();
        playTask.execute(file.getAbsolutePath());
    }

    /**
     *播放任务
```

```
     */
    private class PlayTask extends AsyncTask<String, Void, Exception> {
        /**
         *后台播放任务
         * @param file
         *WAVE 文件
         */
        protected Exception doInBackground(String... file) {
            Exception result = null;

            try {
                //播放 WAVE 文件
                play(file[0]);
            } catch (IOException ex) {
                result = ex;
            }

            return result;
        }

        /**
         * 执行 Post.
         *
         * @param ex
         * 异常实例.
         */
        protected void onPostExecute(Exception ex) {
            //如果播放失败则显示错误信息
            if (ex != null) {
                new AlertDialog.Builder(MainActivity.this)
                        .setTitle(R.string.error_alert_title)
                        .setMessage(ex.getMessage()).show();
            }
        }
    }

    /**
     *
     使用原生 API 播放指定的 WAVE 文件
     *
     * @param fileName
     * 文件名.
     * @throws IOException
     */
    private native void play(String fileName) throws IOException;

    static {
        System.loadLibrary("WAVPlayer");
    }
}
```

现在 WAVE 播放器应用程序的 Java 部分准备好了。现在需要开始实现原生 Play 按钮以通过 OpenSL ES 库来播放指定的 WAVE Audio 文件。

13.2.4 实现 WAVE Aduio 播放

如前所述，我们实现了 WAVE 音频播放器应用程序的原生部分，建立了程序的 Java 部分并确保其可通过编译。按以下步骤将实现播放功能。

(1) 使用 Project Explorer，选定 MainActivity.java 源文件并从主菜单栏中选择 Run | External Tools | Generate C and C++ header file 以生成 com_apress_wavplayer_MainActivity.h 头文件，该头文件声明了原生函数。

(2) 需要修改项目的 Android.mk 构建脚本，以便通过与 wavelib_static 库文件静态链接来获得对 WAVE 文件格式的支持，并通过与 OpenSL ES 库文件动态链接来使用 OpenSL ES 原生音频 API。在 Eclipse 中打开构建脚本，参照程序清单 13-5 替换其内容。

程序清单 13-5　jni/Android.mk 构建脚本内容

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := WAVPlayer
LOCAL_SRC_FILES := WAVPlayer.cpp

# 使用 WAVLib 静态库
LOCAL_STATIC_LIBRARIES += wavlib_static

# 与 OpenSL ES 链接
LOCAL_LDLIBS += -lOpenSLES

include $(BUILD_SHARED_LIBRARY)

# 引入 WAVLib 库模块
$(call import-module, transcode-1.1.5/avilib)
```

(3) 在 Elipse 中打开 WAVPlayer.cpp 原生资源文件。开始部分包含必要的头文件以使用 OpenSL ES API 和 WAVLib API，如程序清单 13-6 所示。

程序清单 13-6　jni/WAVPlayer.cpp 源文件中包含的头文件

```
#include "com_apress_wavplayer_MainActivity.h"

#include <SLES/OpenSLES.h>
#include <SLES/OpenSLES_Android.h>

extern "C" {
#include <wavlib.h>
}
```

```
static const char* JAVA_LANG_IOEXCEPTION = "java/lang/IOException";
static const char* JAVA_LANG_OUTOFMEMORYERROR =
       "java/lang/OutOfMemoryError";

#define ARRAY_LEN(a) (sizeof(a) / sizeof(a[0]))
```

(4) OpenSL ES 原生音频 API 被设计为以异步方式运行。在整个播放过程中，OpenSL ES 引擎将调用一个指定的回调函数来提供音频数据。该函数需要访问播放器上下文以呈现其功能。PlayContext 结构体用来给已注册的回调函数提供播放器上下文。PlayerContext 保存有 OpenSL ES、WAVLib 结构和音频缓冲区。附上 PlayerContext 的源文件 WAVPlayer.cpp 如程序清单 13-7 所示。

程序清单 13-7　PlayerContext 结构持有 Native Context

```
/**
 * Player context.
 */
struct PlayerContext
{
    SLObjectItf engineObject;
    SLEngineItf engineEngine;
    SLObjectItf outputMixObject;
    SLObjectItf audioPlayerObject;
    SLAndroidSimpleBufferQueueItf audioPlayerBufferQueue;
    SLPlayItf audioPlayerPlay;
    WAV wav;

    unsigned char* buffer;
    size_t bufferSize;

    PlayerContext()
    : engineObject(0)
    , engineEngine(0)
    , outputMixObject(0)
    , audioPlayerBufferQueue(0)
    , audioPlayerPlay(0)
    , wav(0)
    , bufferSize(0)
    {}
};
```

(5) ThrowException 作为一个辅助函数，用来在发生错误时方便地将异常抛出至 Java 层。附上这些函数的源文件，如程序清单 13-8 所示。

程序清单 13-8　ThrowException 辅助函数

```
/**
 *从给定的类和消息中抛出异常
 *
```

```
 * @param env JNIEnv interface.
 * @param className class name.
 * @param message exception message.
 */
static void ThrowException(
        JNIEnv* env,
        const char* className,
        const char* message)
{
    //获取异常类
    jclass clazz = env->FindClass(className);

    //如果异常类被发现
    if (0 != clazz)
    {
        // 抛出异常
        env->ThrowNew(clazz, message);

        // 释放类引用
        env->DeleteLocalRef(clazz);
    }
}
```

(6) OpenWaveFile 函数打开指定的 WAVE 音频文件，CloseWaveFile 函数会在不需要该文件时将其释放。如果发生错误，这两个函数均会抛出 IOException 通知 Java 应用程序，并弹出警告对话框来提醒用户。附上这些函数的源文件，如程序清单 13-9 所示。

程序清单 13-9　打开和关闭 WAVE 文件的 WAVLib 辅助函数

```
/**
 *打开给定的 WAVE 文件
 *
 * @param env JNIEnv interface.
 * @param fileName file name.
 * @return WAV file.
 * @throws IOException
 */
static WAV OpenWaveFile(
        JNIEnv* env,
        jstring fileName)
{
    WAVError error = WAV_SUCCESS;
    WAV wav = 0;

    //以 C 字符串形式获取文件名
    const char* cFileName = env->GetStringUTFChars(fileName, 0);
    if (0 == cFileName)
        goto exit;

    //打开 WAVE 文件
```

```
    wav = wav_open(cFileName, WAV_READ, &error);

    //释放文件名
    env->ReleaseStringUTFChars(fileName, cFileName);

    //错误检查
    if (0 == wav)
    {
        ThrowException(env,
                JAVA_LANG_IOEXCEPTION,
                wav_strerror(error));
    }

exit:
    return wav;
}

/**
 *关闭给定的 WAVE 文件
 *
 * @param wav WAV file.
 * @throws IOException
 */
static void CloseWaveFile(
       WAV wav)
{
    if (0 != wav)
    {
        wav_close(wav);
    }
}
```

(7) OpenSL ES 函数调用会因为多种原因而失败。每一个 OpenSL ES 函数调用会返回 SLresult 类型的结果码。OpenSL ES 并未提供任何函数将这些结果码转换为可读信息。ResultToString 辅助函数填补了该空白。它获取结果码并返回相应的错误信息。附上 ResultToString 函数，如程序清单 13-10 所示。

程序清单 13-10　转换结果码的 ResultToString 辅助函数

```
/**
 *
 *将 OpenSL ES 结果转换为字符串

 * @param result result code.
 * @return result string.
 */
static const char* ResultToString(SLresult result)
{
    const char* str = 0;
```

```
switch (result)
{
case SL_RESULT_SUCCESS:
    str = "Success";
    break;

case SL_RESULT_PRECONDITIONS_VIOLATED:
    str = "Preconditions violated";
    break;

case SL_RESULT_PARAMETER_INVALID:
    str = "Parameter invalid";
    break;

case SL_RESULT_MEMORY_FAILURE:
    str = "Memory failure";
    break;

case SL_RESULT_RESOURCE_ERROR:
    str = "Resource error";
    break;

case SL_RESULT_RESOURCE_LOST:
    str = "Resource lost";
    break;

case SL_RESULT_IO_ERROR:
    str = "IO error";
    break;

case SL_RESULT_BUFFER_INSUFFICIENT:
    str = "Buffer insufficient";
    break;

case SL_RESULT_CONTENT_CORRUPTED:
    str = "Success";
    break;

case SL_RESULT_CONTENT_UNSUPPORTED:
    str = "Content unsupported";
    break;

case SL_RESULT_CONTENT_NOT_FOUND:
    str = "Content not found";
    break;

case SL_RESULT_PERMISSION_DENIED:
    str = "Permission denied";
    break;
```

```
        case SL_RESULT_FEATURE_UNSUPPORTED:
            str = "Feature unsupported";
            break;

        case SL_RESULT_INTERNAL_ERROR:
            str = "Internal error";
            break;

        case SL_RESULT_UNKNOWN_ERROR:
            str = "Unknown error";
            break;

        case SL_RESULT_OPERATION_ABORTED:
            str = "Operation aborted";
            break;

        case SL_RESULT_CONTROL_LOST:
            str = "Control lost";
            break;

        default:
            str = "Unknown code";
        }

        return str;
    }
```

(8) CheckError 辅助函数在结果码指示一个相应的错误时抛出一个 IOException 异常。它使用 ResultToString 函数把对应结果码转换为一个信息。附上 CheckError 函数，如程序清单 13-11 所示。

程序清单 13-11　CheckError 函数会在有错误的时候抛出一个异常

```
/**
 *检查结果是否出错，并抛出含错误信息的 IOException。

 * @param env JNIEnv interface.
 * @param result result code.
 * @return error occurred.
 * @throws IOException
 */
static bool CheckError(
        JNIEnv* env,
        SLresult result)
{
    bool isError = false;

    //如果发生错误
    if (SL_RESULT_SUCCESS != result)
    {
```

```
            // 抛出 IOException
            ThrowException(env,
                    JAVA_LANG_IOEXCEPTION,
                    ResultToString(result));
            isError = true;
        }
        return isError;
    }
```

(9) 虽然 OpenSL ES API 是基于 C 语言的，但它采用面向对象的方法实现。每个 OpenSL ES 的结构体建立在两个主要结构之上：对象和接口。对象是针对定义明确的任务所指定的抽象资源集合。接口是对象提供的相关功能的抽象集合。对象可以对外提供一个或多个接口，对象可通过引擎对象或对象接口创建。每个 OpenSL ES 应用程序应首先先创建一个引擎对象，以便可以访问其他 API 函数。引擎通过 slCreateEngine API 创建。CreateEngine 辅助功能用该函数创建引擎对象并在其失败时抛出 IOException 异常。附加 CreateEngine 函数的源代码，如程序清单 13-12 所示：

程序清单 13-12　创建引擎对象的 CreateEngine 函数

```
/**
 *
 *创建 OpenSL ES 引擎

 * @param env JNIEnv interface.
 * @param engineObject object to hold engine. [OUT]
 * @throws IOException
 */
static void CreateEngine(
        JNIEnv* env,
        SLObjectItf& engineObject)
{
    //
    // Android 中 OpenSL ES 被设计为是线程安全的，
    // 所以该选项请求将被忽略，但它应保证源代码可被移植到其他平台
    SLEngineOption engineOptions[] = {
        { (SLuint32) SL_ENGINEOPTION_THREADSAFE,
          (SLuint32) SL_BOOLEAN_TRUE }
    };

    //创建 OpenSL ES 引擎对象
    SLresult result = slCreateEngine(
            &engineObject,
            ARRAY_LEN(engineOptions),
            engineOptions,
            0, //没有接口
            0, //没有接口
            0); //无须提供
    //错误检查
    CheckError(env, result);
}
```

(10) 对象一经创建，便处于未实例化状态，对象虽已存在但未分配任何资源。为使用该对象须先将其实例化。可通过 Object 接口提供的 Realize 函数来完成。RealizeObject 辅助函数用来实现对象并在自身执行失败时抛出一个 IOException 异常。附件该函数的源文件，如程序清单 13-13 所示。

程序清单 13-13　实现对象实例 RealizeObject 函数

```
/**
 *
 *实现给定的对象，Object 在使用前应该先被实现。

 * @param env JNIEnv interface.
 * @param object object instance.
 * @throws IOException
 */
static void RealizeObject(
        JNIEnv* env,
        SLObjectItf object)
{
    //实现该引擎对象
    SLresult result = (*object)->Realize(
            object,
            SL_BOOLEAN_FALSE); // No async, blocking call

    //错误检查
    CheckError(env, result);
}
```

(11) 当不再需要对象时，要将其销毁以释放所分配的资源。可以通过 Object 接口提供的 Destroy 函数实现。附上 DestroyObject 函数的源代码，如程序清单 13-14 所示。

程序清单 13-14　DestroyObject 函数用来销毁不再使用的对象

```
/**
 *
 *销毁给定的对象实例

 * @param object object instance. [IN/OUT]
 */
static void DestroyObject(SLObjectItf& object)
{
    if (0 != object)
        (*object)->Destroy(object);

    object = 0;
}
```

(12) 每一个对象可以有一个或多个接口。这些接口可通过 Object 接口提供的 GetInterface 函数获得。该 GetEngineInterface 辅助函数从给定的 Engine Object 中获得 Engine Interface。

附上该函数的源文件，内容如程序清单 13-15 所示。

程序清单 13-15　获得 Engine Interface 的 GetEngineInterface 函数

```
/**
 *
 *从给定的引擎对象中获得引擎接口以便从该引擎中创建其他对象。

 * @param env JNIEnv interface.
 * @param engineObject engine object.
 * @param engineEngine engine interface. [OUT]
 * @throws IOException
 */
static void GetEngineInterface(
        JNIEnv* env,
        SLObjectItf& engineObject,
        SLEngineItf& engineEngine)
{
    //获得引擎接口
    SLresult result = (*engineObject)->GetInterface(
            engineObject,
            SL_IID_ENGINE,
            &engineEngine);

    // 误检查
    CheckError(env, result);
}
```

(13) CreateOutputMix 函数通过调用带有一组参数的 Engine Interface 的 CreateOutputMix 函数来创建 Output Mixer 对象。附上 CreateOutputMix 函数的源文件，内容如程序清单 13-16 所示：

程序清单 13-16　创建 Output Mixer 的 CreateOutputMix 函数

```
/**
 *
 *创建和输出混合对象

 * @param env JNIEnv interface.
 * @param engineEngine engine engine.
 * @param outputMixObject object to hold the output mix. [OUT]
 * @throws IOException
 */
static void CreateOutputMix(
        JNIEnv* env,
        SLEngineItf engineEngine,
        SLObjectItf& outputMixObject)
{
    //创建输出混合对象
    SLresult result = (*engineEngine)->CreateOutputMix(
```

```
                engineEngine,
                &outputMixObject,
                0, // 没有接口
                0, // 没有接口
                0); // 无须提供

        //错误检查
        CheckError(env, result);
    }
```

(14) InitPlayerBuffer 辅助函数创建一个字节缓冲区来保存音频数据块，并在不需要该缓冲区时用 FreePlayerBuffer 将其释放。InitPlayerBuffer 函数通过 WAVE 音频文件头获取根占用缓冲区值的大小，该值根据输入文件的大小得出。附上该函数的源代码，如程序清单 13-17 所示：

程序清单 13-17　InitPlayerBuffer 和 FreePlayerBuffer 辅助函数

```
/**
 *
 *释放播放器缓冲区

 * @param buffers buffer instance. [OUT]
 */
static void FreePlayerBuffer(unsigned char*& buffers)
{
    if (0 != buffers)
    {
        delete buffers;
        buffers = 0;
    }
}

/**
 *
 *初始化播放器缓冲区

 * @param env JNIEnv interface.
 * @param wav WAVE file.
 * @param buffers buffer instance. [OUT]
 * @param bufferSize buffer size. [OUT]
 */
static void InitPlayerBuffer(
        JNIEnv* env,
        WAV wav,
        unsigned char*& buffer,
        size_t& bufferSize)
{
    //计算缓冲区大小
    bufferSize = wav_get_channels(wav) * wav_get_rate(wav)
                * wav_get_bits(wav);
```

```
        //初始化 buffer
        buffer = new unsigned char[bufferSize];
        if (0 == buffer)
        {
            ThrowException(env,
                    JAVA_LANG_OUTOFMEMORYERROR,
                    "buffer");
        }
    }
```

(15) 为了通过 OpenSL ES 播放 WAVE 音频文件，需要使用带有缓冲区队列的音频播放器。CreateBufferQueueAudioPlayer 函数创建一个简单的 Android 缓冲区队列，该缓冲区队列只有一个 buffers slots 作为音频源。为提高质量，可以适当选择使用更多的 buffers slots。该函数根据 WAVE 音频文件头的结果来定义 PCM 播放的参数。音频播放器的输出方式被设置为 Output Mixer。附上该函数，内容如程序清单 13-18 所示：

程序清单 13-18　CreateBufferQueueAudioPlayer 函数

```
/**
 *创建缓冲区队列音频播放器。
 *
 * @param wav WAVE file.
 * @param engineEngine engine interface.
 * @param outputMixObject output mix.
 * @param audioPlayerObject audio player. [OUT]
 * @throws IOException
 */
static void CreateBufferQueueAudioPlayer(
        WAV wav,
        SLEngineItf engineEngine,
        SLObjectItf outputMixObject,
        SLObjectItf& audioPlayerObject)
{
    // Android 针对数据源的简单缓冲区队列定位器
    SLDataLocator_AndroidSimpleBufferQueue dataSourceLocator = {
        SL_DATALOCATOR_ANDROIDSIMPLEBUFFERQUEUE, // 定位器类型
        1 // 缓冲区数
    };

    // PCM 数据源格式
        SLDataFormat_PCM dataSourceFormat = {
        SL_DATAFORMAT_PCM, // 格式类型
        wav_get_channels(wav), // 通道数
        wav_get_rate(wav) * 1000, //毫赫兹/秒的样本数
        wav_get_bits(wav), // 每个样本的位数
        wav_get_bits(wav), // 容器大小
        SL_SPEAKER_FRONT_CENTER, // 通道屏蔽
        SL_BYTEORDER_LITTLEENDIAN // 字节顺序
```

```
    };

    //数据源是含有 PCM 格式的简单缓冲区队列
    SLDataSource dataSource = {
        &dataSourceLocator, // 数据定位器
        &dataSourceFormat // 数据格式
    };

    //针对数据接收器的输出混合定位器
    SLDataLocator_OutputMix dataSinkLocator = {
        SL_DATALOCATOR_OUTPUTMIX, // 定位器类型
        outputMixObject //输出混合
    };

    //数据定位器是一个输出混合
    SLDataSink dataSink = {
        &dataSinkLocator, // 定位器
        0 // 格式
    };

    //需要的接口
    SLInterfaceID interfaceIds[] = {
        SL_IID_BUFFERQUEUE
    };

    //需要的接口。如果所需要的接口不要用，请求将失败。
    SLboolean requiredInterfaces[] = {
        SL_BOOLEAN_TRUE // for SL_IID_BUFFERQUEUE
    };

    //创建音频播放器对象
    SLresult result = (*engineEngine)->CreateAudioPlayer(
            engineEngine,
            &audioPlayerObject,
            &dataSource,
            &dataSink,
            ARRAY_LEN(interfaceIds),
            interfaceIds,
            requiredInterfaces);
}
```

(16) 通过 Buffer Queue Interface 对缓冲区进行管理。通过该接口，对缓冲区进行排序播放，而且通过注册一个回调函数，在队列中的缓冲区被音频播放器播放完时可收到通知。附上该函数的源文件，如程序清单 13-19 所示。

程序清单 13-19 GetAudioPlayerBufferQueueInterface 函数

```
/**
 *获得音频播放器缓冲区队列接口
 *
```

```
 * @param env JNIEnv interface.
 * @param audioPlayerObject audio player object instance.
 * @param audioPlayerBufferQueue audio player buffer queue. [OUT]
 * @throws IOException
 */
static void GetAudioPlayerBufferQueueInterface(
        JNIEnv* env,
        SLObjectItf audioPlayerObject,
        SLAndroidSimpleBufferQueueItf& audioPlayerBufferQueue)
{
    //获得缓冲区队列接口
    SLresult result = (*audioPlayerObject)->GetInterface(
            audioPlayerObject,
            SL_IID_BUFFERQUEUE,
            &audioPlayerBufferQueue);

    // 错误检查
    CheckError(env, result);
}
```

(17) DestroyContext 函数用于在播放器停止时释放 OpenSL ES 资源和缓冲区。附上该函数，如程序清单 13-20 所示。

程序清单 13-20　释放播放器上下文的 DestroyContext 函数

```
/**
 *销毁播放器上下文
 *
 * @param ctx player context.
 */
static void DestroyContext(PlayerContext*& ctx)
{
    //销毁音频播放器对象
    DestroyObject(ctx->audioPlayerObject);

    //释放播放器缓冲区
    FreePlayerBuffer(ctx->buffer);

    //销毁输出混合对象
    DestroyObject(ctx->outputMixObject);

    //销毁引擎实例
    DestroyObject(ctx->engineObject);

    //关闭 WAVE 文件
    CloseWaveFile(ctx->wav);

    //释放上下文
    delete ctx;
    ctx = 0;
}
```

(18) 当播放器完成对前一个缓冲区队列的播放时，OpenSL ES 音频播放器对象会调用 PlayerCallback 函数。在该回调中，应用程序简单地读取和排列下一个用于播放的音频数据块。如果 WAVE 音频文件播放结束，DestroyContext 函数会被调用以释放资源。附上该函数的源代码，如程序清单 13-21 所示。

程序清单 13-21　PlayerCallback 函数

```
/**
 *当一个缓冲区完成播放时获得调用
 *
 * @param audioPlayerBufferQueue audio player buffer queue.
 * @param context player context.
 */
static void PlayerCallback(
        SLAndroidSimpleBufferQueueItf audioPlayerBufferQueue,
        void* context)
{
    //获得播放器上下文
    PlayerContext* ctx = (PlayerContext*) context;

    //读取数据
    ssize_t readSize = wav_read_data(
            ctx->wav,
            ctx->buffer,
            ctx->bufferSize);

    //如果数据被读取
    if (0 < readSize)
    {
        (*audioPlayerBufferQueue)->Enqueue(
                audioPlayerBufferQueue,
                ctx->buffer,
                readSize);
    }
    else
    {
        DestroyContext(ctx);
    }
}
```

(19) PlayerCallback 通过 Buffer Queue Interface 提供的 RegisterCallback 函数进行注册。在注册过程中，会提供一个上下文指针，以便当音频播放器调用它时该回调函数可以收到这个上下文指针。附上该函数的源代码，如程序清单 13-22 所示。

程序清单 13-22　RegisterPlayerCallback 函数

```
/**
 *注册播放器回调
 *
```

```
 * @param env JNIEnv interface.
 * @param audioPlayerBufferQueue audio player buffer queue.
 * @param ctx player context.
 * @throws IOException
 */
static void RegisterPlayerCallback(
        JNIEnv* env,
        SLAndroidSimpleBufferQueueItf audioPlayerBufferQueue,
        PlayerContext* ctx)
{
    //注册播放器回调
    SLresult result = (*audioPlayerBufferQueue)->RegisterCallback(
            audioPlayerBufferQueue,
            PlayerCallback,
            ctx); // player context

    //错误检查
    CheckError(env, result);
}
```

(20) Play Interface 用来与音频播放器交互。GetAudioPlayerPlayInterface 辅助函数从给定的 Audio Player Object 中获得 Play Interface，如程序清单 13-23 所示。

程序清单 13-23 GetAudioPlayerPlayInterface 函数

```
/**
 *获得音频播放器播放接口
 *
 * @param env JNIEnv interface.
 * @param audioPlayerObject audio player object instance.
 * @param audioPlayerPlay play interface. [OUT]
 * @throws IOException
 */
static void GetAudioPlayerPlayInterface(
        JNIEnv* env,
        SLObjectItf audioPlayerObject,
        SLPlayItf& audioPlayerPlay)
{
    //获得播放器接口
    SLresult result = (*audioPlayerObject)->GetInterface(
            audioPlayerObject,
            SL_IID_PLAY,
            &audioPlayerPlay);

    //错误检查
    CheckError(env, result);
}
```

(21) 音频播放器可以通过用 Play Interface 提供的 SetPlayState 函数来启动。一旦播放器被设置为播放状态，该音频播放器开始等待缓冲区排队就绪。SetAudioPlayerStatePlaying

函数将音频播放器设置为播放状态，如程序清单 13-24 所示。

程序清单 13-24　SetAudioPlayerStatePlaying 函数

```
/**
 *把音频播放器设置为播放状态
 *
 * @param env JNIEnv interface.
 * @param audioPlayerPlay play interface.
 * @throws IOException
 */
static void SetAudioPlayerStatePlaying(
        JNIEnv* env,
        SLPlayItf audioPlayerPlay)
{
    //把音频播放器状态设置为播放
    SLresult result = (*audioPlayerPlay)->SetPlayState(
            audioPlayerPlay,
            SL_PLAYSTATE_PLAYING);

    // 错误检查
    CheckError(env, result);
}
```

(22) 现在所有函数已经准备好了，原生播放函数用前面已经实现的辅助函数来实现播放器流程，如程序清单 13-25 所示。

程序清单 13-25　Play Native Method 实现播放器逻辑

```
void Java_com_apress_wavplayer_MainActivity_play(
        JNIEnv* env,
        jobject obj,
        jstring fileName)
{
    PlayerContext* ctx = new PlayerContext();

    //打开 WAVE 文件
    ctx->wav = OpenWaveFile(env, fileName);
    if (0 != env->ExceptionOccurred())
        goto exit;

    //创建 OpenSL ES 引擎
    CreateEngine(env, ctx->engineObject);
    if (0 != env->ExceptionOccurred())
        goto exit;

    //实现引擎对象
    RealizeObject(env, ctx->engineObject);
    if (0 != env->ExceptionOccurred())
        goto exit;
```

```
//获得引擎接口
GetEngineInterface(
        env,
        ctx->engineObject,
        ctx->engineEngine);
if (0 != env->ExceptionOccurred())
    goto exit;

//创建输出混合对象
CreateOutputMix(
        env,
        ctx->engineEngine,
        ctx->outputMixObject);
if (0 != env->ExceptionOccurred())
    goto exit;

//实现输出混合对象
RealizeObject(env, ctx->outputMixObject);
if (0 != env->ExceptionOccurred())
    goto exit;

// 初始化缓冲区
InitPlayerBuffer(
        env,
        ctx->wav,
        ctx->buffer,
        ctx->bufferSize);
if (0 != env->ExceptionOccurred())
    goto exit;

//创建缓冲区队列音频播放器对象
CreateBufferQueueAudioPlayer(
        ctx->wav,
        ctx->engineEngine,
        ctx->outputMixObject,
        ctx->audioPlayerObject);
if (0 != env->ExceptionOccurred())
    goto exit;

//实现音频播放器对象
RealizeObject(env, ctx->audioPlayerObject);
if (0 != env->ExceptionOccurred())
    goto exit;

//获得音频播放器缓冲区队列接口
GetAudioPlayerBufferQueueInterface(
        env,
        ctx->audioPlayerObject,
        ctx->audioPlayerBufferQueue);
```

```
    if (0 != env->ExceptionOccurred())
        goto exit;

    //注册播放器回调函数
    RegisterPlayerCallback(
            env,
            ctx->audioPlayerBufferQueue,
            ctx);
    if (0 != env->ExceptionOccurred())
        goto exit;

    //获得音频播放器播放接口
    GetAudioPlayerPlayInterface(
            env,
            ctx->audioPlayerObject,
            ctx->audioPlayerPlay);
    if (0 != env->ExceptionOccurred())
        goto exit;

    //设置音频播放器为播放状态
    SetAudioPlayerStatePlaying(env, ctx->audioPlayerPlay);
    if (0 != env->ExceptionOccurred())
        goto exit;

    //将第一个缓冲区入队来启动运行
    PlayerCallback(ctx->audioPlayerBufferQueue, ctx);
    exit:
        //如果发生异常就销毁
        if (0 != env->ExceptionOccurred())
            DestroyContext(ctx);
}
```

将应用程序和原生模块实现一起重新构建之后，就可以准备体验示例应用程序了。

13.3　运行 WAVE Audio Player

为了体验基于 OpenSL ES 的 WAVE 播放器，按照以下步骤来运行该应用程序。

(1) 在运行该应用程序之前，需要有一个 WAVE 音频文件样本。用你的浏览器从 www.nch.com.au/acm/8k16bitpcm.wav 处下载 8 000Hz 16 位 PCM WAVE 音频文件样本。

(2) 用 ADB 调用下面的命令：将 WAVE 音频文件放在目标设备的 SD 卡或模拟器中。

```
adb push 8k16bitpcm.wav /sdcard/
```

(3) 现在启动该应用程序。

(4) 该应用程序启动之后，一个简单 GUI 将会显示出来，如图 13-1 所示。

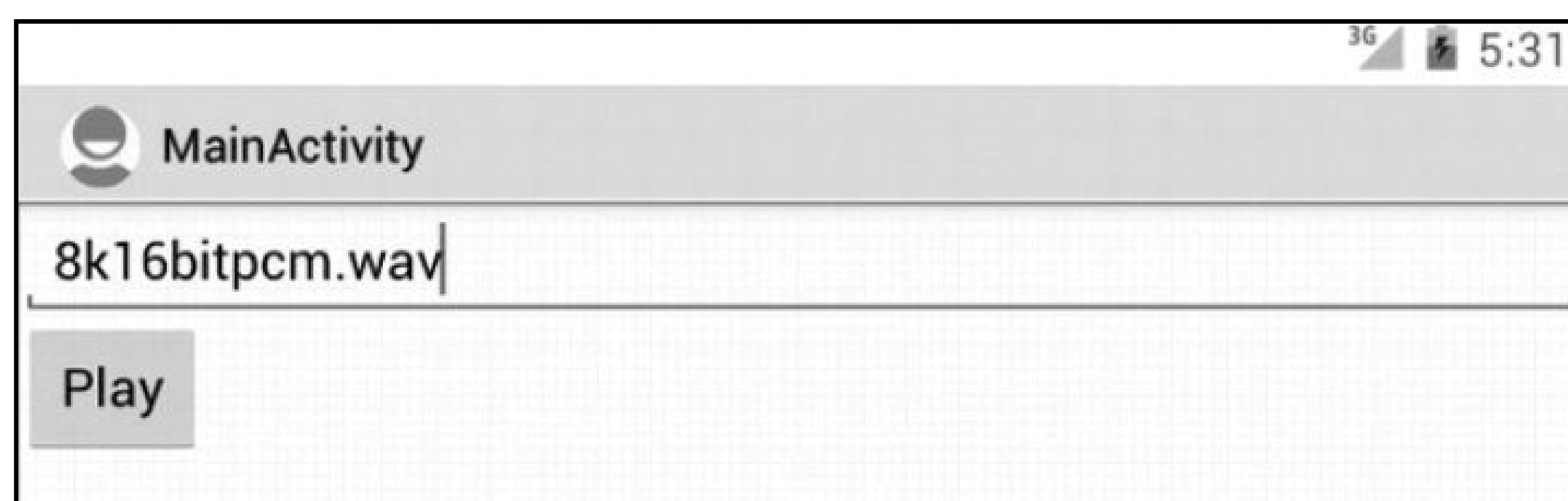


图 13-1　WAVE 播放器简单的用户界面

(5) 单击 Play 按钮启动该播放器，WAVE 音频文件将开始播放。

13.4　小结

本章学习了 Android 平台为原生代码提供的 OpenSL ES 原生音频 API。通过使用这个 API，原生代码能够播放和录制音频而无须与 Java 层进行通信。该功能极大地提高了多媒体应用程序的性能。

第14章

程序概要分析和 NEON 优化

在前面几章中，我们学习了如何在 Android 平台上开发原生应用程序，介绍了 Android 平台和 Linux 操作系统提供的原生 API。最后一章将涵盖以下主要内容：

- 使用 GNU Profiler 剖析原生 Android 应用程序以确定性能瓶颈。
- 通过内部编译器使用 ARM NEON 技术优化原生应用程序。
- 在编译器内启用自动向量化支持，在不改变源代码的情况下无缝地提高原生应用程序的性能。

14.1　用 GNU Profiler 度量性能

GNU Profiler 也被称为 gprof 应用程序，是基于 UNIX 的程序概要分析工具。gprof 可以通过使用仪器和抽样收集和报告每个函数消耗的绝对执行时间。如果编译时提供了-pg 操作，仪器可以通过 GNU C/C++编译器来执行。程序执行后，抽样数据会自动地保存在 gmon.out 数据文件里，之后 gprof 工具会通过处理该数据文件来生成程序概要分析报告。Android NDK 也配有 gprof 工具。然而 GNU C/++编译器工具链配备的 Android NDK 缺少__gnu_mcount_nc 函数的实现，该函数是测试函数运行消耗时间所必需的。为了在 Android NDK 原生项目中使用 gprof 工具，我们将使用一个名为 Android NDK Profiler 的开源项目。更多关于 Android NDK Profiler 开源项目的信息，可以在其官方网站 http://code.google.com/p/android-ndk-profiler/中找到。

14.1.1　安装 Android NDK Profiler

按照以下步骤安装 Android NDK Profiler 原生模块：

(1) 通过浏览器进入 https://github.com/cinar/android-ndk-profiler/zipball/master 下载 Android NDK Profiler 原生模块 ZIP 格式的文件。

(2) 直接把 ZIP 归档文件中的内容提取到 NDK 原生模块子目录 ANDROID_NDK_HOME/

source 中。将提取出的目录 cinar-android-ndk-profiler-9cdf13 重命名为 android-ndk-profiler。

现在，作为一个原生模块，Android NDK Profiler 已经就绪，任何 Android NDK 原生项目都可以使用它。现在马上就要学习启用 Android NDK Profiler 以便原生项目可以使用它。

14.1.2 启用 Android NDK Profiler

需要在编译期间启用 Android NDK Profiler 以便收集程序概要分析的数据。为了在原生 Android 项目中启用 Android NDK Profiler，请按照下列步骤配置。

(1) 需要修改 Android.mk 生成脚本，以实现同之前安装的 andprof 库进行静态链接。可以按照程序清单 14-1 所示修改 Android.mk 文件。

程序清单 14-1 在 Android.mk 生成脚本中启用 Android NDK Profiler

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := module

...


#启用 Android NDK Profiler
MY_ANDROID_NDK_PROFILER_ENABLED := true

#如果 Android NDK Profiler 已经被启用
ifeq ($(MY_ANDROID_NDK_PROFILER_ENABLED),true)

#显示信息
$(info GNU Profiler is enabled)

#启用监控功能
LOCAL_CFLAGS += -DMY_ANDROID_NDK_PROFILER_ENABLED

#使用 Enable the monitor functions 静态库
LOCAL_STATIC_LIBRARIES += andprof
Endif

...

include $(BUILD_SHARED_LIBRARY)

...

#如果 Android NDK Profiler 已经被启用
ifeq ($(MY_ANDROID_NDK_PROFILER_ENABLED),true)
# 导入 Android NDK Profiler 库模块
```

```
$(call import-module, android-ndk-profiler/jni)
endif
```

(2) 根据这些修改，可以通过将构建系统变量 MY_ANDROID_NDK_PROFILER_ENABLED 设置为 true 或 false 来启用和禁用程序概要分析。

(3) 因为原生代码运行在共享库内，所以程序概要分析生命周期需要手动地进行管理。Android NDK Profiler 提供函数来启动和停止收集程序概要分析数据。这些函数已经在 prof.h 头文件中进行了声明，所以在使用这些函数之前，首先要将 prof.h 头文件包含进来，如程序清单 14-2 所示：

程序清单 14-2　将 Android NDK Profiler 头文件包含进来

```
#ifdef MY_ANDROID_NDK_PROFILER_ENABLED
#include <prof.h>
#endif
```

(4) 为了开始收集程序概要分析数据，需要调用 monstartup 函数。monstartup 函数获取共享库的名字作为其参数并开始收集程序概要分析数据。根据应用程序的生命周期，在你想要的点调用 monstartup 函数和收集程序概要分析数据。如程序清单 14-3 所示：

程序清单 14-3　通过调用 monstartup 函数来启动数据收集

```
#ifdef MY_ANDROID_NDK_PROFILER_ENABLED
    //启动收集示例
    monstartup("libModule.so");
#endif
```

(5) 通过调用 moncleanup 函数来停止收集程序概要分析数据。moncleanup 函数负责将收集的程序概要分析数据保存到 SD 卡上的 gmon.out 文件里。如程序清单 14-4 所示：

程序清单 14-4　调用 moncleanup 来停止收集数据

```
#ifdef MY_ANDROID_NDK_PROFILER_ENABLED
    // 存储已收集的数据
    moncleanup();
#endif
```

注意：

在对应用程序进行程序概要分析之前，请确保应用程序有适当的权限可以对 SD 卡进行写操作。

14.1.3　使用 GNU Profiler 分析 gmon.out 文件

可以使用 GNU Profiler gprof 工具对 Android NDK Profiler 生成的程序概要分析数据文件 gmon.out 进行处理。该工具会根据提供的程序概要分析数据文件生成可读的报告。请按照下面这些步骤来处理 gmon.out 文件。

(1) 使用 adb 命令将程序概要分析数据文件 gmon.out 从 SD 卡上下载到计算机上。

```
adb pull /sdcard/gmon.out
```

(2) GNU Profiler 需要调试标志和程序概要分析数据文件来生成报告。启动 arm-linux-androideabi-gprof.exe 应用程序，该应用程序带有共享库的调试版本和 gmon.out 程序概要分析数据文件。

```
%ANDROID_NDK_HOME%\toolchains\arm-linux-androideabi-4.4.3\prebuilt\
windows\bin\armlinux-
    androideabi-gprof.exe obj\local\armeabi-v7a\libModule.so gmon.out
```

(3) 用 arm-linuxandroideabi-gprof 基于主机平台中的正确位置来替换应用程序的路径。用正在对其进行程序概要分析文件的正确体系结构来替换 armeabi-v7a 目录。

(4) GNU Profiler 将会解析程序概要分析数据文件和生成报告，如程序清单 14-5 所示。生成的报告有两部分组成，flat profile 和 call graph。两个部分都用表格来呈现程序概要分析数据，报告里还提供了对每个测试的描述。

程序清单 14-5　GNU Profiler 报告文件

```
Flat profile:

Each sample counts as 0.01 seconds.
  % cumulative self self total
time seconds seconds calls ms/call ms/call name
99.53 2.12 2.12 361 5.87 5.87 func2
 0.47 2.13 0.01 func1

...

                      Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.47% of 2.13 seconds

index % time self children called name
                                     <spontaneous>
[1] 99.5 0.00 2.12          func1 [1]
         2.12 0.00 361/361      func2 [2]
-------------------------------------
```

当实现了新的功能或者对应用程序进行了优化时，就可以重复这些步骤来监控应用程序的性能。在第 14.2 节中，当通过 ARM NEON　intrinsics 优化原生函数时将会用到 GNU Profiler。

14.2　使用 ARM NEON Intrinsics 进行优化

在本节中，我们将会重新使用在 12 章实现的基于 Bitmap renderer 的 AVI 播放器示例应用程序。并通过使用纯 C 代码实现亮度过滤器来扩展该示例应用程序。在本节的后面部

分，将会重新实现亮度过滤器函数，并使用 ARM NEON intrinsics 优化其性能。并会像前面章节中描述的那样，我们将使用 GNU Profiler 比较两个实现的性能。

14.2.1　ARM NEON 技术概述

在 ARM 处理器中实现的单指令、多数据(single instruction，multiple data，SLMD)被称为 NEON。SIMD 通过多个数据点上执行相同的操作实现数据级并行。SIMD 技术可以通过启用单指令向量运算，来对原生应用程序性能进行加速。多媒体应用程序从 SIMD 技术受益最多，因为它们可以对多种格式的数据集执行相同的操作，比如视频帧或音频数据块。NEON 技术适用于大部分的 ARM Cortex-A 系列处理器。

在 NEON 技术中，数据被组织成 64 位 D 寄存器或 128 位的 Q 寄存器。这些寄存器可以容纳 8 位、16 位、32 位和 64 位宽的数据向量，如图 14-1 所示。

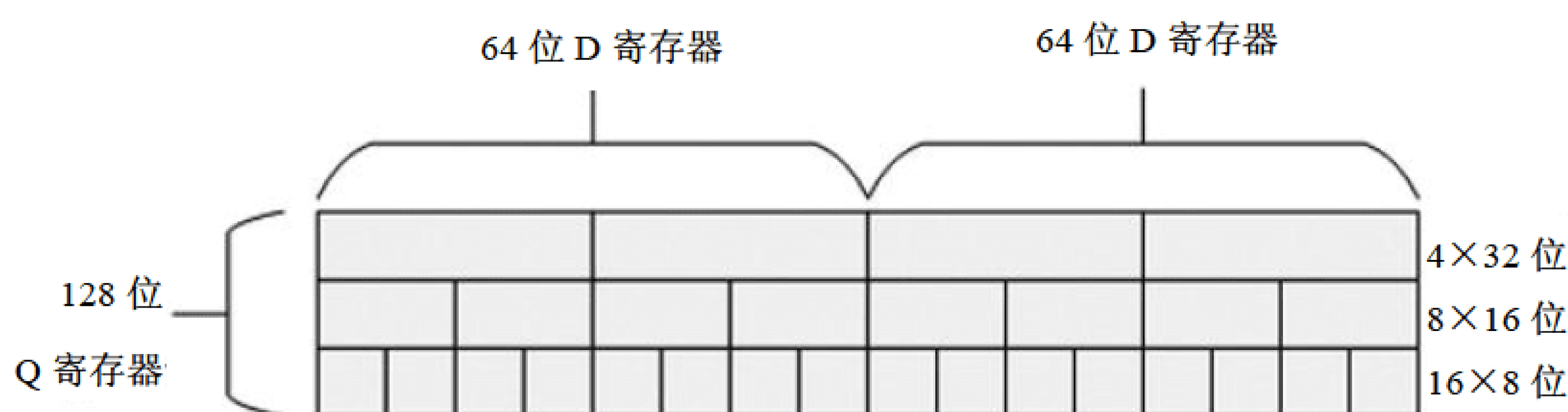


图 14-1　NEON 寄存器和数据类型

NEON 技术同样提供了一套指令集来操作这些数据向量。更多关于 NEON 技术以及其支持的指令等信息可以在 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/ch01s04s03.html 中 ARM 的“Introducing NEON Development Article”中找到。

14.2.2　给 AVI Player 添加一个亮度过滤器

按照以下步骤添加亮度播放器。

(1) 使用 Project Explorer，在 jni 子目录下创建一个新的 C/C++头文件。

(2) 将 C/C++头文件命名为 BrightnessFilter.h 并用程序清单 14-6 中的代码修改文件中的内容。

程序清单 14-6　BrightnessFilter.h 头文件内容

```
#pragma once

/**
 * 提取交错部分，RGB565 色彩空间总共有 16 位
 * 其中 红色 5 位，绿色 6 位和蓝色 5 位
 */
void brightnessFilter(
        unsigned short* pixels,
        long count,
        unsigned char brightness);
```

(3) 创建新的 C/C++资源文件，并命名为 BrightnessFilter.cpp，然后修改其内容，如程序清单 14-7 所示。brightnessFilter 函数只是简单地调用 genericBrightnessFilter 函数，该函数是使用纯 C 实现了亮度过滤器的功能。它采用一个 16 位像素的数组来格式化使用 RGB565 色彩空间。它将颜色组成部分分解成为三个 8 位的值，并根据所给的亮度值给他们增量。同样，该函数会根据每个值的范围对其进行校正，并将它们结合到一起放入一个 RGB565 色彩空间内的 16 位像素中。

程序清单 14-7　源文件 BrightnessFilter.cpp 的内容

```
#include "BrightnessFilter.h"

static void genericBrightnessFilter(
        unsigned short* pixels,
        long count,
        unsigned char brightness)
{
    const unsigned char MAX_RB = 0xF8;
    const unsigned char MAX_G = 0xFC;

    unsigned short r, g, b;

    for (long i = 0; i < count; i++)
    {
        //分解颜色
        r = (pixels[i] >> 8) & MAX_RB;
        g = (pixels[i] >> 3) & MAX_G;
        b = (pixels[i] << 3) & MAX_RB;

        //亮度增量
        r += brightness;
        g += brightness;
        b += brightness;

        //确保颜色组成部分的值在范围内
        r = (r > MAX_RB) ? MAX_RB : r;
        g = (g > MAX_G) ? MAX_G : g;
        b = (b > MAX_RB) ? MAX_RB : b;

        //设置像素
        pixels[i] = (r << 8);
        pixels[i] |= (g << 3);
        pixels[i] |= (b >> 3);
    }
}

void brightnessFilter(
        unsigned short* pixels,
        long count,
        unsigned char brightness)
```

```
{
    genericBrightnessFilter(pixels, count, brightness);
}
```

(4) 每个 AVI 视频帧渲染之前都需要调用 Brightness Filter。为此，打开 com_apress_aviplayer_BitmapPlayerActivity.cpp 源文件。

(5) 将 BrightnessFilter.h 头文件添加到包含列表中，如程序清单 14-8 所示。

程序清单 14-8　将 BrightnessFilter.h 头文件添加到 BitmapRenderer 中

```
extern "C" {
#include <avilib.h>
}

#include <android/bitmap.h>
#include "BrightnessFilter.h"
#include "Common.h"
#include "com_apress_aviplayer_BitmapPlayerActivity.h"

...
```

(6) 修改 renderer 函数如程序清单 14-9 所示，这样它处理每一帧画面时都会调用 brightnessFilter 函数。

程序清单 14-9　对每一帧调用 brightnessFilter 函数

```
jboolean Java_com_apress_aviplayer_BitmapPlayerActivity_render(
        JNIEnv* env,
        jclass clazz,
        jlong avi,
        jobject bitmap)
{
    ...

    //将 AVI 帧字节读入到 bitmap
    frameSize = AVI_read_frame((avi_t*) avi, frameBuffer, &keyFrame);

    // 应用亮度过滤器
    brightnessFilter((unsigned short*) frameBuffer, frameSize/2, 1);

    ...
}
```

(7) 将 BrightnessFilter.cpp 源文件加入到 Android.mk 构建脚本中，如程序清单 14-10 所示。

程序清单 14-10　将 BrightnessFilter.cpp 源文件添加到 Android.mk 中

```
LOCAL_PATH := $(call my-dir)
```

```
include $(CLEAR_VARS)

LOCAL_MODULE := AVIPlayer
LOCAL_SRC_FILES := \
    Common.cpp \
    com_apress_aviplayer_AbstractPlayerActivity.cpp \
    com_apress_aviplayer_BitmapPlayerActivity.cpp

LOCAL_SRC_FILES += BrightnessFilter.cpp

# 使用 AVILib 静态库
LOCAL_STATIC_LIBRARIES += avilib_static
```

(8) 现在 Brightness Filter 已经整合到 AVI Player 应用程序中。在启动应用程序之前，先要启用 GNU Profiler。

14.2.3 为 AVI 播放器启用 Android NDK Profiler

如前所述，为了收集程序概要分析数据，需要在编译时启用 GNU Profiler。

按照以下步骤为 Bitmap renderer AVI Player 启用 GNU Profiler。

(1) 修改 Android.mk 构建脚本来启用 GNU Profiler。

(2) 使用 Project Explorer 展开 jni 子目录，并打开源文件 com_apress_aviplayer_AbstractPlayerActivity.cpp。

(3) 修改代码来调用 Android NDK Profiler 函数，如程序清单 14-11 所示。当 AVI 打开和关闭时，程序概要分析就会相应地开始和结束。这提供了 AVI 处理期间的程序概要分析数据。

程序清单 14-11 从 AbstractPlayerActivity 中调用 Profiler 函数

```
extern "C" {
#include <avilib.h>
}

#ifdef MY_ANDROID_NDK_PROFILER_ENABLED
#include <prof.h>
#endif

...

jlong Java_com_apress_aviplayer_AbstractPlayerActivity_open(
        JNIEnv* env,
        jclass clazz,
        jstring fileName)
{
    avi_t* avi = 0;

#ifdef MY_ANDROID_NDK_PROFILER_ENABLED
    //开始收集样本
```

```
    monstartup("libAVIPlayer.so");
#endif

    ...
}

...

void Java_com_apress_aviplayer_AbstractPlayerActivity_close(
        JNIEnv* env,
        jclass clazz,
        jlong avi)
{
    AVI_close((avi_t*) avi);

#ifdef MY_ANDROID_NDK_PROFILER_ENABLED
    // 存储所收集的数据
    moncleanup();
#endif
}
```

(4) 因为 Android NDK Profiler 将程序概要分析数据文件存储到 SD 卡上，需要给清单文件赋予恰当的授权。用 Project Explorer 打开 AndroidManifest.xml 文件，如程序清单 14-12 所示对其进行修改。

程序清单 14-12　添加外部存储器的写入权限

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.apress.aviplayer"
    android:versionCode="1"
    android:versionName="1.0" >

    ...

    <uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

    ...

</manifest>
```

现在对于 AVI Player 项目来说 GNU Profiler 是可用的了。可以启动应用程序来收集程序概要分析数据了。

14.2.4　AVI Player 程序概要分析

按照以下步骤来对 Bitmap renderer AVI Player 应用程序进行概要分析。

(1) 启动一个实际 Android 设备上的应用程序。

(2) 用 Bitmap renderer 开始播放 AVI 文件。

(3) 一直等到 AVI 播放结束。

(4) 单击设备上的硬返回键。

(5) 正如本章前面所说明的，从设备上将 gmon.out 程序概要分析数据下载到计算机中。

(6) 如程序清单 14-13 所示，使用 gprof 工具生成报告。

程序清单 14-13　通用亮度过滤器程序概要分析报告

```
Flat profile:

Each sample counts as 0.01 seconds.
  % cumulative self self total
 time seconds seconds calls ms/call ms/call name
100.00 2.62 2.62 361 7.26 7.26 brightnessFilter(unsigned short*,
long, unsigned char)
```

根据这份报告，brightnessFilter 函数调用 genericBrightnessFilter 函数处理每帧画面花费 7.26 微秒，而处理所有帧总共花费 2.62 秒。

14.2.5　使用 NEON Intrinsics 优化 Brightness Filter

现在将要使用 ARM NEON intrinsics 优化 genericBrightnessFilter 函数。

(1) 使用 Project Explorer 定位到 jni 子目录。

(2) 打开 BrightnessFilter.cpp 源文件，并添加 NEON 优化过的 neonBrightnessFilter 函数，如程序清单 14-14 所示。相对于通用的亮度过滤器的实现，ARM NEON 优化过的亮度过滤器一次能够处理 8 个像素而不是只处理 1 个像素。

程序清单 14-14　修改后的 BrightnessFilter.cpp 源文件内容

```
#include "BrightnessFilter.h"

#ifdef __ARM_NEON__

#include <cpu-features.h>

#include <arm_neon.h>

static void neonBrightnessFilter(
        unsigned short* pixels,
        long count,
        unsigned char brightness)
{
    const unsigned char MAX_RB = 0xF8;
    const unsigned char MAX_G = 0xFC;

    uint8x8_t maxRb = vmov_n_u8(MAX_RB);
    uint8x8_t maxG = vmov_n_u8(MAX_G);
    uint8x8_t increment = vmov_n_u8(brightness);
```

```
for (long i = 0; i < count; i += 8)
{
    //加载 8 个 16 位像素
    uint16x8_t rgb = vld1q_u16(&pixels[i]);

    // r = (pixels[i] >> 8) & MAX_RB;
    uint8x8_t r = vshrn_n_u16(rgb, 8);
    r = vand_u8(r, maxRb);

    // g = (pixels[i] >> 3) & MAX_G;
    uint8x8_t g = vshrn_n_u16(rgb, 3);
    g = vand_u8(g, maxG);

    // b = (pixels[i] << 3) & MAX_RB;
    uint8x8_t b = vmovn_u16(rgb);
    b = vshl_n_u8(b, 3);
    b = vand_u8(b, maxRb);

    // r += brightness;
    r = vadd_u8(r, increment);

    // g += brightness;
    g = vadd_u8(g, increment);

    // b += brightness;
    b = vadd_u8(b, increment);

    // r = (r > MAX_RB) ? MAX_RB : r;
    r = vmin_u8(r, maxRb);

    // g = (g > MAX_G) ? MAX_G : g;
    g = vmin_u8(g, maxG);

    // b = (b > MAX_RB) ? MAX_RB : b;
    b = vmin_u8(b, maxRb);

    // pixels[i] = (r << 8);
    rgb = vshll_n_u8(r, 8);

    // pixels[i] |= (g << 3);
    uint16x8_t g16 = vshll_n_u8(g, 8);
    rgb = vsriq_n_u16(rgb, g16, 5);

    // pixels[i] |= (b >> 3);
    uint16x8_t b16 = vshll_n_u8(b, 8);
    rgb = vsriq_n_u16(rgb, b16, 11);

    // 存储 8 16 位像素
    vst1q_u16(&pixels[i], rgb);
}
```

```
}

#endif

static void genericBrightnessFilter(
        unsigned short* pixels,
        long count,
        unsigned char brightness)
```

(3) 当适用时，同样需要修改 brightnessFilter 函数以便调用 NEON 优化过的函数。ARM NEON 支持仅对 armeabi v7a ABI 有效。但是请注意，并不是每个基于 ARM-v7 的设备都支持 NEON 指令。原生应用程序对基于 ARM-v7 的设备都希望检测 NEON 支持。为了解决这一问题，Android NDK 自带了 CPU Features 原生导入模块。该模块可以在运行时检测 CPU 的型号和 CUP 支持的功能。修改 brightnessFilter 函数，如程序清单 14-15 所示。

注意：

不是每一个基于 ARM-v7 的设备都支持 ARM NEON 指令。所以应该在运行时调用任何 NEON 优化过的函数之前，总要先使用 CPU Features 导入模块来检测 NEON 支持。

程序清单 14-15　修改 brightnessFilter 函数调用 NEON 优化过的函数

```
void brightnessFilter(
        unsigned short* pixels,
        long count,
        unsigned char brightness)
{
#ifdef __ARM_NEON__

    //获取 CPU 系列
    AndroidCpuFamily cpuFamily = android_getCpuFamily();

    //获取 CPU 功能
    uint64_t cpuFeatures = android_getCpuFeatures();

    //仅在支持 NEON 的 ARM CPUs 上使用 NEON 优化函数
    if ((ANDROID_CPU_FAMILY_ARM == cpuFamily)
            && ((ANDROID_CPU_ARM_FEATURE_NEON & cpuFeatures) != 0))
    {
        //调用 NEON 优化亮度过滤器
        neonBrightnessFilter(pixels, count, brightness);
    }
    else
    {
#endif
        //调用通用的亮度过滤器
        genericBrightnessFilter(pixels, count, brightness);
#ifdef __ARM_NEON__
    }
#endif
}
```

(4) 打开 Android.mk 构建脚本并对其进行修改，如程序清单 14-16 所示。这样就可以在编译期间编译出合适版本的 brightnessFilter 函数。针对 ARMv7a 目标平台，会使用 NEON 加强版本的 brightnessFilter。对于所有其他的平台，会使用 brightnessFilter 的通用实现。

程序清单 14-16　将 brightnessFilter 的 NEON 版本添加到 Android.mk

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := AVIPlayer
LOCAL_SRC_FILES := \
    Common.cpp \
    com_apress_aviplayer_AbstractPlayerActivity.cpp \
    com_apress_aviplayer_BitmapPlayerActivity.cpp

# 添加 armeabi-v7a 上的 NEON 优化版本
ifeq ($(TARGET_ARCH_ABI),armeabi-v7a)
    LOCAL_SRC_FILES += BrightnessFilter.cpp.neon
    LOCAL_STATIC_LIBRARIES += cpufeatures
else
    LOCAL_SRC_FILES += BrightnessFilter.cpp
endif

#使用 AVILib 静态库
LOCAL_STATIC_LIBRARIES += avilib_static

# 启用 Android NDK Profiler
MY_ANDROID_NDK_PROFILER_ENABLED := true

# Android NDK Profiler 是否启用
ifeq ($(MY_ANDROID_NDK_PROFILER_ENABLED),true)

#显示信息
$(info GNU Profiler is enabled)

#启用监控功能
LOCAL_CFLAGS += -DMY_ANDROID_NDK_PROFILER_ENABLED

#使用 Enable the monitor functions 静态库
LOCAL_STATIC_LIBRARIES += andprof
endif

#链接 JNI graphics
LOCAL_LDLIBS += -ljnigraphics

include $(BUILD_SHARED_LIBRARY)

#导入 AVILib 库模块
$(call import-module, transcode-1.1.5/avilib)
```

```
#如果 Android NDK Profiler 已经被启用
ifdef MY_ANDROID_NDK_PROFILER_ENABLED
#导入 Android NDK Profiler 库模块
$(call import-module, android-ndk-profiler/jni)
endif

# 在 armeabi-v7a 上添加 CPU 功能
ifeq ($(TARGET_ARCH_ABI),armeabi-v7a)
# 引入 Android CPU 功能
$(call import-module, android/cpufeatures)
endif
```

注意：

你可能已经注意到 BrightnessFilter.cpp 源文件已经追加了.neon 后缀。这个后缀告诉 Android NDK 构建系统该源文件需要同 ARM NEON 支持一起编译。

(5) 创建一个名为 Application.mk 并包含下面内容的新文件：

```
APP_ABI := armeabi-v7a
```

(6) 当要对 NEON-enhanced brightnessFilter 做程序概要分析时，更好的做法是将 armeabi-v7a ABI 作为单一的目标平台。

(7) 重复相同的程序概要分析步骤。由 GNU Profiler 生成的报告会和程序清单 14-17 很相似。

程序清单 14-17　NEON 优化的 Brightness Filter 程序概要分析报告

```
Flat profile:

Each sample counts as 0.01 seconds.
  % cumulative self self total
 time seconds seconds calls ms/call ms/call name
100.00 0.50 0.50 361 1.39 1.39 brightnessFilter(unsigned short*,
long, unsigned char)
```

根据这份报告，调用 neonBrightnessFilter 函数的 brightnessFilter 函数处理每帧画面花费 1.39 微秒，而处理所有帧总共花费 0.50 秒。相比于通用的实现，NEON-optimized 函数要快 5 倍。

14.3　自动向量化

正如在第 14.2 节中看到的，采用 ARM NEON 支持对应用程序的性能有很大的影响。但是，无论是用 ARM 汇编语言或 NEON intrinsic 结构进行优化都需要保证流畅性。NEON 是一个 SIMD 在 ARM 上的特定实现版本。为了支持不同于 ARM 平台，比如 Intel 或 MIPS，同样也需要为其他的 SIMD 版本，比如 Intel SSE 或 MIPS MDMX 提供最优化的函数实现。

汇编语言或 intrinsics 不是从 SIMD 支持中获益的唯一途径。得益于 SIMD 引擎,在大多数情况下 GNU C/ C+ +编译器也可以自动优化应用程序，而无须写一行汇编代码或使用 intrinsics。这个过程被称为自动向量化。

14.3.1　启用自动向量化

为了启用自动向量化，请遵照下面这些步骤：

(1) 打开 Application.mk 构建脚本，并确保 APP_ABI 包含 armeabi-v7a。

```
APP_ABI := armeabi armeabi-v7a
```

(2) 打开 Application.mk 构建脚本，并给构建系统变量 LOCAL_CFLAGS 加上参数 -ftree-vectorize，如程序清单 14-18 所示。

程序清单 14-18　启用 GNU C/C++编译器自动向量化

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := module

...

LOCAL_CFLAGS += -ftree-vectorize

...

include $(BUILD_SHARED_LIBRARY)
```

(3) 确保源文件和 ARM NEON 支持一起得到编译，如程序清单 14-19 所示。

程序清单 14-19　对所有的源文件启用 ARM NEON

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

...

# 将 ARM NEON 支持添加到所有源文件中
ifeq ($(TARGET_ARCH_ABI),armeabi-v7a)
LOCAL_ARM_NEON := true
endif

...

include $(BUILD_SHARED_LIBRARY)
```

根据这些修改，GNU C/C++编译器会试图自动向量化原生应用程序，进而从 ARM NEON 支持中受益。

C/C++语言没有提供任何机制来指定并行行为。关于哪些代码可以安全的进行自动向量化，可有需要给 GNU C/C++编译器额外的提示。至于自动向量化循环的一览表，请在 http://gcc.gnu.org/projects/tree-ssa/vectorization.html 地址下参见“Auto-vectorization in GCC”文件。

14.3.2 自动向量化问题的发现和排除

当要发现并排除自动向量化中的问题时，可以通过给构建系统变量 LOCAL_CFLAGS 加上自变量－ftree-vectorizer-verbose=2 从 GNU C/C++编译器请求更详细的输出信息。

```
LOCAL_CFLAGS += -ftree-vectorizer-verbose=2
```

一旦指定该函数，GNU C/C++编译器会产生一个输出信息，如程序清单 14-20 所示。该输出信息将给出编译器如何对待应用程序的每一循环的建议。

程序清单 14-20 自动向量化详细输出

```
Cygwin : Generating dependency file converter script
Compile thumb : Vectorization <= Vectorization.c

jni/Vectorization.c:9: note: not vectorized: complicated access pattern.
jni/Vectorization.c:4: note: vectorized 0 loops in function.

jni/Vectorization.c:28: note: LOOP VECTORIZED.
jni/Vectorization.c:22: note: LOOP VECTORIZED.
jni/Vectorization.c:18: note: vectorized 2 loops in function.
Executable : Vectorization
Install : Vectorization => libs/armeabi-v7a/Vectorization
```

根据编译器的详细输出信息来调整代码，从而给编译器提供关于应用程序中每一个循环的恰当的提示。

14.4 小结

本章学习了怎样使用 Android NDK Profiler 库和 GNU Profiler 应用程序来分析原生应用程序。同时也探讨了怎样使用 ARM NEON 技术来优化原生应用程序的性能。